



EFFECTIVE MULTI-THREADING IN OS/2°



Len Dorfman Marc J. Neuberger



Effective Multithreading in OS/2®

Effective Multithreading in OS/2®

Len Dorfman Marc J. Neuberger

McGraw-Hill, Inc.

New York San Francisco Washington, D.C. Auckland Bogotá
Caracas Lisbon London Madrid Mexico City Milan
Montreal New Delhi San Juan Singapore
Sydney Tokyo Toronto

OS/2 Accredited logo is a trademark of IBM Corp. and is used by McGraw-Hill, Inc. under license. Effective Multithreading in OS/2 is independently published by McGraw-Hill, Inc. IBM Corp. is not responsible in any way for the contents of this book.

McGraw-Hill, Inc. is an accredited member of the IBM Independent Vendor League.

Library of Congress Cataloging-in-Publication Data Dorfman, Len.

Effective multithreading in OS/2 / by Len Dorfman and Marc J.

Neuberger.

p. cm. Includes index.

ISBN 0-07-017841-0 (pbk.)

1. Operating systems (Computers) 2. OS/2 (Computer file)

QA76.76.063D6694 1994

005.4'469-dc20

93-38300

CIP

Copyright © 1994 by McGraw-Hill. Printed in the United States of America. Except as permitted under the United States Copyright Act of 1976, no part of this publication may be reproduced or distributed in any form or by any means, or stored in a data base or retrieval system, without the prior written permission of the publishers.

1 2 3 4 5 6 7 8 9 0 DOH/DOH 9 9 8 7 6 5 4

ISBN 0-07-017841-0

The sponsoring editor for this book was Jennifer Holt DiGiovanna, the editor was John C. Baker, the designer was Jaclyn J. Boone, and the production supervisor was Katherine G. Brown. This book was set in Century Schoolbook. It was composed by TAB Books.

Printed and bound by R. R. Donnelley of Harrisonburg.

Information contained in this work has been obtained by McGraw-Hill, Inc. from sources believed to be reliable. However, neither McGraw-Hill nor its authors guarantees the accuracy or completeness of any information published herein and neither McGraw-Hill nor its authors shall be responsible for any errors, omissions, or damages arising out of use of this information. This work is published with the understanding that McGraw-Hill and its authors are supplying information but are not attempting to render engineering or other professional services. If such services are required, the assistance of an appropriate professional should be sought.

For more information about other McGraw-Hill materials, call 1-800-2-MCGRAW in the United States. In other countries, call your nearest McGraw-Hill office.

To Barbara and Rachel, for their love and spirited humor. To Stephen Moore for his eagle's eye.

Len

To Liz, for being there during the preparation of this book, and putting up with Len's epic telephone messages.

Marc

Contents

Part 1 Multithreading Background			
Chapter 1. Multithreading Concepts	3		
Single vs. Multitasking	3		
Multithreading			
Summary	6		
Chapter 2. Programming with Threads	7		
Race Conditions	7		
Critical Sections			
Mutual Exclusion	13		
Synchronization Busy Waiting	16 17		
Event Semaphores			
Message Queues			
Deadlocks			
Summary	29		
Part 2 OS/2 Multithreading Facilities			
Chapter 3. Starting and Ending Threads	33		
Summary	41		
_beginthread()	42		
_endthread()	43		
_threadstore()	44		
DosCreateThread() DosKillThread()	45		
DosWaitThread()	47 48		
20011411111044()	40		

Introduction xi Overview xi

What You Need to Use This Book xii

Building the Programs xii

Chapter 4. Enabling and Disabling Thread Rescheduling	49
Summary	49
DosEnterCritSec()	52
DosExitCritSec()	53
Chapter 5. Suspending and Resuming Thread Execution	55
Summary	60
DosSuspendThread()	61
DosResumeThread()	62
Chapter 6. Changing the Priority of a Thread	63
Summary	68
DosSetPriority()	69
Chapter 7. Mutual Exclusion Semaphores	71
Summary	83
DosCloseMutexSem()	84
DosCreateMutexSem()	85
DosOpenMutexSem()	87
DosQueryMutexSem()	88
DosReleaseMutexSem()	89
DosRequestMutexSem(90
Chapter 8. Event Semaphores	91
Summary	97
DosCloseEventSem()	98
DosCreateEventSem()	99
DosOpenEventSem()	100
DosPostEventSem()	101
DosQueryEventSem()	102
DosResetEventSem()	103
DosWaitEventSem()	104
Chapter 9. Multiple Wait Semaphores	105
Summary	117
DosAddMuxWaitSem()	118
DosCloseMuxWaitSem()	119
DosCreateMuxWaitSem()	120
DosDeleteMuxWaitSem()	122
DosOpenMuxWaitSem()	123
DosQueryMuxWaitSem()	124
DosWaitMuxWaitSem()	125
Chapter 10. Using Queues	127
Summary	133
DosCloseQueue()	134
DosCreateQueue()	135
DosOpenQueue()	136

DosPeekQueue()	137			
DosPurgeQueue()	139			
DosReadQueue()	140 142			
DosQueryQueue()				
DosWriteQueue()	143			
Chapter 11. Using Timers	145			
Summary	149			
DosAsyncTimer()	150			
DosStartTimer()	151			
DosStopTimer()	152			
Part 3 Real Multithreading Programs				
Chapter 12. A Simple Problem, Many Incorrect Solutions	155			
Summary	180			
01 10 71 7 7 7				
Chapter 13. The Producer/Consumer Problem	181			
Summary	199			
Chapter 14. A Complex Example	201			
Chapter 14. A Complex Example	201			
Summary	212			
Chapter 15. Event-Driven Input	213			
Cummony	222			
Summary	222			
Part 4 Multithreading and the Presentation Manager				
Chapter 16. Using Server Threads under PM	225			
Summary	253			
Chapter 17. Multithreaded Painting	255			
Summary	274			
Cummuny	214			
Chapter 18. Using Multiple Message Queue threads	275			
Summary	284			
Enilogue 205				
Epilogue 285 Index 287				
About the Authors 289				

Introduction

This book is not intended to be a cookbook for writing multithreaded programs. While numerous techniques are illustrated that can be used in a wide variety of situations, every application is different and will likely require unique multithreading techniques. Multithreading is hard work. As soon as the number of threads in a program exceeds one, a whole array of potential bugs presents itself. The aim of this book is to raise your level of sophistication with respect to multithreading. There are some 40 multithreaded programs in this book. Studying these programs along with the text will make you multithreading-savvy. You will understand the standard multithreading techniques and be aware of many of the traps and pitfalls that await you in the world of multithreading.

Although writing multithreaded programs is much harder than writing conventional programs, it often is unavoidable. There are many situations, particularly in Presentation Manager applications, where you must use multithreading to make your application user-friendly.

The book's three primary goals are:

- Present the basic concepts of writing multithreaded programs.
- Familiarize the reader with the OS/2 multithreading API.
- Show, through the use of examples, how to write reliable multithreaded programs in OS/2.

Overview

We begin in Part 1 by giving some background on the history and uses of multithreading. Many OS/2 programmers come from platforms in which multithreading does not exist, for example DOS, Windows, and most flavors of UNIX. Part 1 will be extremely useful to these people. Readers familiar with multithreading concepts will find Part 1 to contain some useful ideas, although most of it will likely be review.

Part 2 starts the discussion of OS/2 multithreading by presenting an overview of the OS/2 services that relate to multithreading. While this chapter is no substitute for IBM's excellent OS/2 2.0 Technical Reference Library, Control Program Programming Reference, and Programming Guide Volume 1, it will be particularly useful to the programmer who lacks these reference works. This part of the book

presents numerous instructive sample programs that demonstrate the multithreading services. The example programs form the backbone of this book. We will rely heavily on examples to illustrate the points that we are making.

The examples in Part 2 are largely aimed at demonstrating the multithreading services and pointing out their subtleties. In Part 3, we will begin to present practical multithreading programming techniques. We will use as an example a multithreaded copy program. We will show several techniques for interfacing between tasks to speed a file copying operation. We also will present a program that uses multithreading to achieve an event-driven programming model in a character-mode application.

Finally, in Part 4, we will discuss multithreading and the Presentation Manager. Most OS/2 programmers doing new program development probably are writing Presentation Manager applications. This part will point out the pitfalls and demonstrate techniques for writing multithreaded PM applications.

What You Need to Use This Book

The examples in this book were developed using the IBM CSet++ development environment. We expect that they will run with little modification under Borland C++ for OS/2.

While OS/2 runs on machines with as little as 4M, we strongly recommend having at least 8M of main memory for compilation using the IBM compiler.

Building the Programs

To build or rebuild the sample programs in this book, we have included a make file, cset.mak. To use it, type:

nmake -f cset.mak

Multithreading Background

Chapter

Multithreading Concepts

In this chapter, we will give a brief overview of the history of multithreading and the ideas that underlie it. This will help to understand how and why multithreading is used. It also will be useful to understand how multithreading is implemented.

Single vs. Multitasking

At the dawn of civilization, in the 1950s, when computers were starting to move out of the laboratory and into the real world, they did one thing at a time. One program executed, and another program could not be started until the first had finished. When a program needed to access an I/O device like a card reader or a printer, the whole system waited for the I/O to complete.

It was quickly recognized that keeping the CPU effectively idle for the duration of an I/O operation was a waste of a precious resource. (If you think you paid a lot for your computer, you should price some of the early models.) Why not allow program B to run on the CPU while program A was waiting for I/O to complete? From this desire was born multiprogramming or multitasking.

Multitasking allows many programs to be run concurrently on a single CPU. How does this work? The first thing that you need is an operating system to negotiate the use of the hardware between the various programs that are running. The part of the operating system that decides which program to run next and runs it is called the *scheduler*.

Before discussing the operation of a scheduler, we need to define what a process is. A *process* is simply a running program. The term *program* generally is used to refer to the source code or executable machine code. Program refers to a static, unchanging entity, whereas process refers to the program as it is running.

Just what constitutes a process? The main components of a process are an address space and a CPU state. The *address space* is an area of the system's memory that is accessible to the program. Generally, this area is in some way protected from use by other

4 Multithreading Background

processes. This area, at least initially, will contain an image of the program that the process is running. As the process executes, it modifies data in memory, and thus, in time, the process memory deviates from the program's memory image.

In addition to an address space, a *CPU state* needs to be associated with a process. As the process executes, the registers of the CPU change. One particularly important register is the *program counter*, or PC, which sometimes is referred to as the *instruction pointer*, or IP. This register gives the address of the next instruction to execute in the process. The registers are part of the state of the process. In addition to the address space and CPU state, a process also generally has associated with it open I/O channels and other operating system resources.

Scheduling is the act of picking a process, making its address space available, loading its CPU state registers into the CPU and resuming the process at its next instruction. There are two major classes of scheduling: preemptive and nonpreemptive. A nonpreemptive scheduler, as the name implies, does not preempt one process to run another. A process only loses control of the CPU (stops running to allow another process to run) when it surrenders control to the scheduler. This surrender of control might be as a result of an explicit call to the scheduler from the process, or it might be a result of an I/O operation for which the process must wait.

A preemptive scheduler can take control from a process at any time. This prevents an uncooperative process from monopolizing the CPU. A preemptive scheduler uses timer interrupts to gain control from the process. It then can either resume that process or pass control to another process. Processes that require the CPU are called *ready-to-run*. Some processes are waiting for an external event such as I/O to complete or another process to free a resource. These processes are called *blocked*, *pended*, or *waiting*. These terms will be used interchangeably in this book.

Most operating systems do preemptive scheduling, so we will concentrate on that. A preemptive scheduler can gain control of the CPU in any of several ways:

- The currently running process calls the scheduler explicitly. The currently running process wants to give other processes the chance to run immediately, so it makes a call to the scheduler. The CPU state of the process is saved. Another process is chosen, its CPU state is restored and the new process is resumed.
- The currently running process performs a request that cannot be satisfied immediately. The currently running process makes a request for I/O of some sort (for example, input from the user). The results of this I/O will not be available for some time, so the process will not need the CPU. The state is saved and a new process is scheduled.
- A timer interrupt gives control to the scheduler. The currently running process has been running on the CPU long enough that a timer interrupt has occurred. The state has been saved as part of the interrupt. The scheduler picks a process and schedules it.

■ An I/O interrupt can give control to the scheduler. An I/O request completes. The completion is signalled by an interrupt (for example, a keyboard interrupt caused by the user's striking a key). The completion of the I/O can make a process ready-to-run. The scheduler picks a process from among the ready-to-run processes and schedules it.

Multithreading

One of the key features of multitasking is that each process has it own address space. Sometimes, however, it would be nice to be able to do several things at once within a process. Consider a program that copies from a hard disk to a diskette (for example, a backup program). The obvious program reads from disk, perhaps does a little processing, then writes to the diskette, then reads from disk, etc. If it takes 1 minute to read all of the data from the hard disk, 5 seconds to do whatever processing is required, and 2 minutes to write it to diskette, then the whole process will take 3 minutes, 5 seconds. However, we're leaving our resources (CPU, hard disk, diskette drive) idle for significant periods. Is there a way we can get higher utilization and thus speed the process?

What we need is overlapped I/O. Overlapped I/O is exactly what it sounds like—doing separate I/O operations at the same time. There are two general ways to achieve this. The first is via unpended I/O. Recall from the previous section our saying that, when a process issues an I/O request, it becomes pended until that request has been satisfied, allowing other processes to run. With unpended I/O, the call returns immediately, the process doesn't pend. Clearly, the call then returns before the I/O has been completed, so the process needs some means to determine when the I/O finally is complete. This generally is accomplished through a pending call that waits for any of the I/O operations issued to complete. It returns some identifier for the request that has completed, and the process proceeds. Managing such an arrangement requires the programmer to do a lot of work and do a lot of bookkeeping.

A more general approach to obtaining overlapped I/O (and the subject of this book) is multithreading. *Threads*, sometimes referred to as *lightweight processes*, are like processes in that they are schedulable entities. However, all of the threads that make up a process share both the same address space and most other resources, such as file I/O channels. Thus, with multithreading, we have the power of doing many things at the same time without the expense of multiple address spaces and interprocess communication. Threads in the same process can communicate through memory, because they share an address space. Each thread in a process has its own stack area. This is not to say that an errant thread can't write to another task's stack, but there is an area of memory set aside for each thread.

6 Multithreading Background

There are two approaches to scheduling in a multithreaded system. One approach is *two-level scheduling*. The scheduler first must choose a process to run, then it chooses a thread within that process. In OS/2, a simpler approach is used: the scheduler chooses threads directly.

Multithreading gives the programmer a tremendous amount of power. Along with power comes responsibility. When you have several threads running at the same time, sharing the same memory, a whole new dimension of bugs arises. This is the topic of our next chapter.

Summary

Multitasking is the ability to run several processes simultaneously on a single system. It provides more efficient use of resources, allowing one process to use the CPU while another is waiting for I/O. Each process has a unique address space. With multithreading, the capability is added to have multiple threads of execution in a single process sharing a single address space.

Chapter

2

Programming with Threads

This chapter outlines some of the basic issues that a programmer must consider when writing a multithreaded program.

Race Conditions

The first basic problem of multithreading can be illustrated with a couple of simple program fragments taken from prog2-1.c, which is presented in Figure 2.1. We will create two tasks each of which increments the same global location.

```
counter;
void thread1(void *x)
 long
                 i;
  long
                 tmp;
  for (i = 0; i < 1000000; i++) {
       tmp= counter;
       tmp++;
       counter= tmp;
}
void thread2(void *x)
 long
                 i;
  long
                 tmp;
  for (i = 0; i < 2000000; i++) {
       tmp= counter;
       tmp++;
       counter= tmp;
}
```

Figure 2.1 prog2-1.c

```
prog2-1.c
        This program demonstrates the basic problem of concurrency.
    Two threads concurrently modify a global location, and mayhem
        Each of the two threads increments a global location a
    set number of times. If the two threads ran serially, the value
   of the counter would be the sum of the two increments at the
   end of execution. However, because a thread can be rescheduled
    in the midst of updating the counter, some updates get lost.
#include <stdio.h>
#include <process.h>
#define INCL DOSPROCESS
#include <os2.h>
#include "mt.h"
 ^{\star} Function declarations for the threads.
void thread2(void *);
void thread3(void *);
   Declare our global counter variable. This is the location that
   the two threads will be fighting over.
long
                         counter;
        Main()
void main()
{
    TID
                         thread2Tid;
                         thread3Tid;
    TID
    counter= 0;
    printf("Program starts:\n");
    printf(" counter= %d\n", counter);
printf(" [Threads start..."); fflush(stdout);
     * Start the threads:
    thread2Tid= _beginthread(
                                             /* Address of function
                         thread2,
                                             /* Don't really give stack
                         STACK SIZE,
                                             /* Size of stack needed
                         (PVOID) 0);
                                              /* Message to thread
    thread3Tid= _beginthread(
                                              /* Address of function
                         thread3,
                                             /* Don't really give stack
                         NULL,
                                             /* Size of stack needed
                         STACK_SIZE,
                                             /* Message to thread
                         (PVOID) 0);
    /*
```

```
* Wait for both threads to complete.
    DosWaitThread(&thread2Tid, DCWW_WAIT);
    DosWaitThread(&thread3Tid, DCWW WAIT);
       Now that they're done, let's see what the counter reads.
       If we had called the functions in serial, we would expect the
       result to be 3,000,000. We will find that it is less, because
       some updates have been lost.
    printf("end]\n");
    printf(" counter= %d\n", counter);
        thread2()
        Thread2 increments the global location "counter" 1,000,000
    times.
void thread2(void *foo)
                        i;
    long
    long
                        tmp;
    for (i = 0; i < 1000000; i++) {
        tmp= counter;
        tmp++;
        counter= tmp;
}
        thread3()
        Thread3 increments the global location "counter" 2,000,000
    times.
void thread3(void *foo)
    long
                        i;
    long
                        tmp;
    for (i = 0; i < 2000000; i++) {
        tmp= counter;
        tmp++;
        counter= tmp;
```

If these routines run serially, counter will be bigger by 3,000,000 after both routines have run. We might expect the same behavior when they are two different threads, but we'd be surprised. If you try running program prog2.-1.exe, you'll find that the

number varies each time that you run it. Why? Recall that with preemptive multitasking, a process (or in this case a thread) can be interrupted at any point to run another thread. Suppose thread 1 is interrupted after the statement tmp= counter; and thread 2 is run for a while, after which thread 1 is resumed. We might have the following interleaving of instructions:

Thread 1 gets the value of counter into a local variable, then thread 2 takes over, incrementing counter. When thread 1 resumes, however, it still has an old value of counter saved in tmp, and it is this old value that it increments and stores. Thus, all of the increments that thread 2 did are lost. This type of bug is called a *timing bug* or a *race condition*, indicating that the bug appears only under certain timing conditions. Such bugs can be extraordinarily difficult to find. Because they often are intermittent, they are difficult to debug. Further, even when a timing bug is reproducible, using a debugger might alter the timing of the program sufficiently that the bug no longer occurs.

What our example has shown us is that the sequence:

```
tmp= counter;
tmp++;
counter= tmp;
```

is not atomic. An *atomic operation* is one that runs from start to finish without anything else happening. Atomicity is relative to how we define "anything else happening." In absolute terms, the only things that are truly atomic are the quantum state transitions of fundamental particles; however, this assumes a rather broad definition of "anything else happening." It generally is useful to limit the scope to the CPU that we're running on. In this context, most instructions are atomic. On some hardware, though, there are even some instructions of sufficient complexity that they can be interrupted in the middle by a hardware interrupt or a page fault. If we replace our three line sequence by the equivalent:

```
counter= counter + 1;
```

the program can begin to work as expected. This is because the code is optimized into one atomic instruction that increments counter. This optimization depends on the compiler, so it might or might not translate into one instruction. A sufficiently brilliant compiler might even be able to optimize the three line sequence into one line, recognizing that the tmp local variable is extraneous.

Critical Sections

However, let's assume that the increment takes several instructions (the safest assumption anyway). Such a sequence of instructions, one which is sensitive to sched-

uling, is called a *critical section*. How do we get our program to work? We must make the critical section atomic relative to other, possibly conflicting, instruction sequences.

The most brute force way to do this is demonstrated in program prog2-2.c, which is presented in Figure 2.2. Here, we surround the critical section with calls to DosEnterCritSec() and DosExitCritSec(). During the period after a call to DosEnterCritSec() and before a call to DosExitCritSec(), no other thread in the same process will be scheduled. The critical section thus is rendered atomic relative to other threads in the same process. The only problem with using this approach is that all other threads in the program are prevented from running. Even threads that do not conflict with the critical section are prevented from running.

The DosEnterCritSec() and DosExitCritSec() calls are discussed more comprehensively in chapter 4.

Figure 2.2 prog2-2.c

```
prog2-2.c
        This program uses DosEnterCritSec() and DosExitCritSec() to
    show how access to global resources can be serialized by
    disabling rescheduling during critical sections of code.
#include <stdio.h>
#include <process.h>
#define INCL_DOSPROCESS
#include <os2.h>
#include "mt.h"
   Function declarations for the threads.
void thread2(void *);
void thread3(void *);
    Declare our global counter variable. This is the location that
   the two threads will be fighting over.
long
                        counter;
        Main()
void main()
    TID
                        thread2Tid;
    TID
                        thread3Tid;
    counter= 0:
    printf("Program starts:\n");
    printf("
               counter= %d\n", counter);
    printf("
                [Threads start..."); fflush(stdout);
```

Figure 2.2 Continued

```
Start the threads:
    thread2Tid= _beginthread(
                                             /* Address of function
                         thread2,
                                             /* Don't really give stack
                        NULL.
                                             /* Size of stack needed
                        STACK SIZE,
                         (PVOID) 0);
                                              /* Message to thread
    thread3Tid= _beginthread(
                                             /* Address of function
                         thread3,
                                             /* Don't really give stack
/* Size of stack needed
/* Message to thread
                         NULL,
                        STACK_SIZE,
                         (PVOID) 0);
        Wait for both threads to complete.
    DosWaitThread(&thread2Tid, DCWW_WAIT);
    DosWaitThread(&thread3Tid, DCWW_WAIT);
     * Now that they're done, let's see what the counter reads.
     * Having used Dos<Enter, Exit>CritSec(), we should get the
     * expected 3,000,000 now.
    printf("end]\n");
    printf(" counter= %d\n", counter);
}
        thread2()
        Thread 2 increments the global location "counter" 1,000,000
    times, protecting the update with DosEnterCritSec().
void thread2(void *foo)
                        i;
    long
    long
                         tmp;
    for (i = 0; i < 1000000; i++) {
         * DosEnterCritSec() will prevent other threads in this
         * process from running, until we call DosExitCritSec().
        DosEnterCritSec();
         * We now have the process to ourselves. No one else can run.
        tmp= counter;
        tmp++;
        counter= tmp;
```

```
We're done accessing the global location, so we can allow
            other threads to run now. DosExitCritSec() does this.
        DosExitCritSec():
}
        thread3()
        Thread 3 increments the global location "counter" 2,000,000
    times, protecting the update with DosEnterCritSec().
void thread3(void *foo)
                        i;
    long
    long
                        tmp;
    for (i = 0; i < 2000000; i++) {
           DosEnterCritSec() will prevent other threads in this
            process from running, until we call DosExitCritSec().
        DosEnterCritSec();
            We now have the process to ourselves. No one else can run.
        tmp= counter;
        tmp++;
        counter= tmp;
            We're done accessing the global location, so we can allow
            other threads to run now. DosExitCritSec() does this.
        DosExitCritSec():
}
```

Mutual Exclusion

What we really want for this particular problem is mutual exclusion. Here, a mutual exclusion, or mutex, semaphore is used to serialize access to the global resource (in this case, the variable counter). Serializing access to a resource means that only one thread at a time will be able to access the resource—in other words, they access the resource in serial. Program prog2-3.c, which is presented in Figure 2.3, shows how mutex semaphores are used to make the program work. Now, the critical section is atomic not to all other tasks in the system but to any other code sequences that conflict with the critical section. Note that all code sequences that access the resource must request and obtain the semaphore before using the resource. If an uncooperative thread fails to obtain the semaphore before accessing the resource, the program might behave incorrectly.

Mutual exclusion semaphores are discussed at length in chapter 7.

Figure 2.3 prog2-3.c

```
prog2-3.c
        This program uses a mutual exclusion semaphore to serialize
    access to a global variable.
#include <stdio.h>
#include <process.h>
#define INCL DOSPROCESS
#define INCL_DOSSEMAPHORES
#include <os2.h>
#include "mt.h"
^{\prime*} ^{\star} Function declarations for the threads.
void thread2(void *);
void thread3(void *);
/*
* Declare our global counter variable:
long
                        counter;
 * Declare a mutex semaphore to protect the variable.
HMTX
                        lockCounter:
        Main()
void main()
                         thread2Tid;
    TID
                        thread3Tid;
        Create our mutex semaphore:
    DosCreateMutexSem(
                                             /* We don't need a name
                                            /* Place to return handle
                        &lockCounter,
                                             /* No options
                                             /* Initially unowned
                         FALSE);
    counter= 0:
    printf("Program starts:\n");
    printf(" counter= %d\n", counter);
               [Threads start..."); fflush(stdout);
    printf("
     * Start the threads:
    thread2Tid= _beginthread(
                                            /* Address of function
                                                                         */
                         thread2,
```

```
NULL.
                                             /* Don't really give stack
                        STACK SIZE,
                                             /* Size of stack needed
                        (PVOID) 0):
                                             /* Message to thread
    thread3Tid= _beginthread(
                        thread3,
                                             /* Address of function
                                            /* Don't really give stack
/* Size of stack needed
                        NULL,
                        STACK_SIZE,
                        (PVOID) 0);
                                            /* Message to thread
        Wait for both threads to complete.
    DosWaitThread(&thread2Tid, DCWW_WAIT);
    DosWaitThread(&thread3Tid, DCWW_WAIT);
       Let's see what the counter reads. It should read the expected
        3,000,000, because we've proteced the updates with a mutual
     * exclusion semaphore.
    printf("end]\n");
    printf(" counter= %d\n", counter);
}
        thread2()
        Thread 2 increments the global location "counter" 1,000,000
    times, protecting the update with a mutual exclusion semaphore.
void thread2(void *foo)
    long
                        i;
    long
                        tmp;
    for (i = 0; i < 1000000; i++) {
        * Before updating the global variable, we need to lock
         * by requesting the mutex semaphore.
        DosRequestMutexSem(
                        lockCounter,
                                                /* Semaphore Handle
                        SEM_INDEFINITE_WAIT); /* Wait till we get it */
           We own the lock now. If anyone else requests it they will
         * block until we release it.
           Therefore, we can do our update now.
         */
        tmp= counter;
        tmp++;
        counter= tmp;
           We're done with the update. Release the lock so someone
           else can get it.
```

```
Figure 2.3 Continued
        DosReleaseMutexSem(lockCounter);
}
        thread3()
        Thread 3 increments the global location "counter" 2,000,000
    times, protecting the update with a mutual exclusion semaphore.
void thread3(void *foo)
{
    long
                        i;
    long
                        tmp;
    for (i = 0; i < 2000000; i++) {
            Before updating the global variable, we need to lock
            by requesting the mutex semaphore.
        DosRequestMutexSem(
                        lockCounter,
                                                 /* Semaphore Handle
                                                /* Wait till we get it */
                        SEM_INDEFINITE_WAIT);
            We own the lock now. If anyone else requests it they will
            block until we release it.
           Therefore, we can do our update now.
        tmp= counter;
        tmp++;
        counter= tmp;
            We're done with the update. Release the lock so someone
            else can get it.
        DosReleaseMutexSem(lockCounter);
}
```

Synchronization

Beyond getting threads to coexist peacefully, it generally is necessary for them to communicate with each other. Perhaps the most important information is information related to timing. It often is necessary for one thread to know where in its execution another thread is. For example, an input event might occur, such as the user clicking a mouse button, that requires one thread to notify another. This problem is known as *synchronization*. There are several ways of communicating this timing information.

Busy Waiting

Because threads in a single process share an address space, they can easily communicate data through memory locations. One way to communicate timing information is through busy-waiting. Program prog2-4.c, which is presented in Figure 2.4, illustrates this technique. Suppose thread 2 needs to wake up when thread 1 gets a mouse click. Thread 2 can execute a loop which checks a flag to see whether thread 1 got a mouse click. When a mouse click occurs, thread 1 sets the flag to true. For example, the following code is from prog2-4.c:

```
void thread1(void *foo)
{
    while (TRUE) {
        /* wait for mouse click */
        mouseClickOccurred= TRUE;
     }
}

void thread2(void *foo)
{
    while (TRUE) {
        while (!mouseClickOccurred) {
            mouseClickOccurred= FALSE;
            }
            ... /* Process mouse click */
     }
}
```

In this code, thread 2 buzzes in a tight loop waiting for mouseClickOccurred to get set to true. While simple, this type of synchronization is highly undesirable because it wastes CPU time. Thread 2 is always ready to run and, therefore, will get its fair share of the CPU time. If these are the only two tasks in the system, thread 2 will get half the CPU while it's doing no useful work.

Figure 2.4 prog2-4.c

```
prog2-4.c

**

This program demonstrates busy-waiting as a synchronization technique.

**

The main thread listens for mouse button 1 clicks. It signals the second thread by incrementing a mouse click counter. The second thread repeatedly looks at the counter to see if it has been incremented. This is a waste of CPU time.

#include <stdio.h>
#include <process.h>

#define INCL_DOSPROCESS
#define INCL_MOU
#include <os2.h>
#include "mt.h"
```

```
Figure 2.4 Continued
```

```
Function declaration for the second thread
void thread2(void *);
* Declare our global counter variable:
long
                        button1Down;
       Main()
void main()
{
   TID
                        thread2Tid;
   HMOU
                       mouHand;
   MOUEVENTINFO
                        mouEvent:
   USHORT
                        fWait;
   USHORT
                        eventMask:
       Set up the mouse so that we can get button events.
   MouOpen(NULL, &mouHand);
   MouDrawPtr(mouHand);
   eventMask= 0x7E;
   MouSetEventMask(&eventMask, mouHand);
       Start the second thread:
    thread2Tid= _beginthread(
                                            /* Address of function
                        thread2.
                                            /* Don't really give stack
                        NULL,
                                            /* Size of stack needed
                        STACK SIZE,
                                            /* Message to thread
                        (PVOID) 0):
   while (TRUE) {
           Wait for a mouse event
       fWait= 1;
       MouReadEventQue(&mouEvent, &fWait, mouHand);
       if (mouEvent.time) {
           /*
* First check for a button 1 press:
            if (mouEvent.fs & MOUSE_BN1_DOWN) {
                /*
* Notify the printing thread. Note the use of
                 * Dos<Enter,Exit>CritSec(). Although button1Down++
                 * probably is an atomic operation, we would prefer not
                 * to assume this, as it could burn us in the future.
                DosEnterCritSec();
                   We use increment here and decrement in the printer
                 * thread on the off chance that two presses will occur
                 * before the other thread is scheduled. While it is
```

```
* unlikely in this case, in other cases where the event
                   of interest is more frequent, it is an important
                   consideration.
                button1Down++;
                DosExitCritSec();
               If button 2 is pressed exit.
            if (mouEvent.fs & MOUSE_BN2_DOWN) {
                break;
    /*
* Let's print something and go.
   printf("done.\n");
        thread2()
       Thread2 checks button1Down to see whether it's set. If so,
    it prints a button 1 down message.
void thread2(void *foo)
       Busy wait looking for button1Down to go nonzero.
    while (TRUE) {
       while (button1Down) {
               We got a button press.
            printf("Button 1 Pressed\n");
               Decrement the counter, indicating that we've serviced
               the button press.
            DosEnterCritSec();
            button1Down";
            DosExitCritSec();
}
```

Event Semaphores

Semaphores are designed to solve this problem. Program prog2-5.c, which is presented in Figure 2.5, illustrates this technique. Thread 2 is coded to wait on a semaphore, and thread 1 to signal it.

```
void thread1(void *foo)
  while (TRUE) {
      /* wait for mouse click */
       DosPostEventSem(mouseClickSem);
void thread2(void *foo)
  while (TRUE) {
       DosWaitEventSem (mouseClickSem,
                      SEM_WAIT_INDEFINITE);
       DosResetEventSem(mouseClickSem,
                       &numClicks);
       ... /* Process mouse click */
}
```

In this program, thread 2 is pended waiting for a signal on mouseClickSem, an event semaphore. When a mouse click occurs, thread 1 signals the semaphore waking up thread 2. Thus, there is no waste of CPU time, because it is not using any CPU while thread 2 is waiting.

Event semaphores are covered in detail in chapter 8.

Figure 2.5 prog2-5.c

```
prog2-5.c
       This program works just like prog2-4.c, except that rather
    than using busy-waiting as a synchronization technique, it
   uses an event semaphore.
        The event semaphore is used to pend the printing task until
    it needs to be woken up. The main thread wakes up the printing
    thread when a mouse button click occurs.
#include <stdio.h>
#include cess.h>
#define INCL_DOSPROCESS
#define INCL_DOSSEMAPHORES
#define INCL_MOU
#include <os2.h>
#include "mt.h"
 * Function declaration for the second thread
void thread2(void *);
 * Declare our semaphore
                        buttonPressedSem:
HEV
```

```
Main()
void main()
   TID
                        thread2Tid;
   HMOU
                        mouHand:
   MOUEVENTINFO
                        mouEvent:
   USHORT
                        fWait;
   USHORT
                        eventMask;
    * Set up the mouse so that we can get button events.
   MouOpen(NULL, &mouHand);
   MouDrawPtr(mouHand);
   eventMask= 0x7E;
   MouSetEventMask(&eventMask, mouHand);
    * Create our event semaphore:
    */
   DosCreateEventSem(
                       NULL,
                                            /* We don't need a name
                                           /* Place to return handle
                       &buttonPressedSem,
                                            /* No options
                        FALSE):
                                            /* Initially unposted
       Start the second thread:
   thread2Tid= _beginthread(
                                            /* Address of function
                        thread2,
                                            /* Don't really give stack
                       NULL,
                                            /* Size of stack needed
                        STACK_SIZE,
                                            /* Message to thread
                        (PVOID) 0);
   while (TRUE) {
        * Wait for a mouse event
       fWait= 1;
       MouReadEventQue(&mouEvent, &fWait, mouHand);
       if (mouEvent.time) {
           /*
* First check for a button 1 press:
           if (mouEvent.fs & MOUSE BN1 DOWN) {
               /*
* Wake the other thread up.
               DosPostEventSem(buttonPressedSem);
              If button 2 is pressed exit.
           if (mouEvent.fs & MOUSE_BN2_DOWN) {
               break:
```

```
Figure 2.5 Continued
```

```
Let's print something and go.
   printf("done.\n");
        thread2()
        Thread2 waits on the event semaphore buttonPressedSem. When
    it is awakened, at least one button press has occurred.
void thread2(void *foo)
   ULONG
                       button1Presses;
       Repeatedly wait for an event then process it.
   while (TRUE) {
           Wait until the other thread pokes us:
       DosWaitEventSem(
                        buttonPressedSem,
                                                /* Semaphore handle
                        SEM INDEFINITE WAIT):
                                                /* Wait forever
           Now, reset the semaphore and see how many times it was posted.
       DosResetEventSem(
                        buttonPressedSem,
                                                /* Semaphore handle
                                                /* # of times posted
                        &button1Presses);
           button1Presses now has the number of times that the semaphore
           was posted. This corresponds to the number of button presses.
       while (button1Presses) {
            * We got a button press.
           printf("Button 1 Pressed\n");
           button1Presses";
        }
}
```

Message Queues

Another, more specific, method of achieving synchronization is through the use of message queues. *Message queues* provide synchronized message passing between

threads. Message queues are appropriate when a server thread is providing a service to one or more client threads. For example, a server thread might provide access to a database. Whenever another thread, a client thread, requires data from the database or needs to update the database, it issues a request to the server. More generally, when a client needs whatever service the server provides, it sends a message to the server, who performs the requested action. In the following code, taken from prog2-6.c, which is presented in Figure 2.6, no useful information is passed in the message, rather it is used merely as a token of a button press:

Message queues are overkill in this example, but it is instructive to see how they can be used to provide synchronization.

Chapter 10 discusses message queues in detail.

Figure 2.6 prog2-6.c

```
prog2-6.c
        This program works just like prog2-5.c, except that rather
    than using an event semaphore for synchronization, it uses a
    aueue.
        The printing task waits for requests on the queue. The main
    task waits for mouse clicks. When a mouse click occurs, it adds
    a request to the queue, causing the printing task to wake up.
    Generally, queues are used when requests contain data. Here, we
    do not actually have any data to pass.
#include <stdio.h>
#include cess.h>
#define INCL_DOSPROCESS
#define INCL_DOSQUEUES
#define INCL MOU
#include <os2.h>
#include "mt.h"
```

Figure 2.6 Continued

```
* Function declaration for the second thread
void thread2(void *);
 * Declare our queue
HQUEUE
                          buttonPressedQ:
        Main()
void main()
{
    TID
                          thread2Tid;
    HMOU
                          mouHand;
                          mouEvent;
    MOUEVENTINFO
    USHORT
                          fWait;
    USHORT
                          eventMask;
     * Set up the mouse so that we can get button events.
    MouOpen(NULL, &mouHand);
    MouDrawPtr(mouHand);
    eventMask= 0x7E;
    MouSetEventMask(&eventMask, mouHand);
       Create the queue that we're going to use for synchronization.
    DosCreateQueue(
                          &buttonPressedQ, /* Address of handle */
0, /* No options */
"\\OUEUES\\prog2-6"); /* Queue name REQUIRED */
        Start the thread
    thread2Tid= _beginthread(
                                                /* Address of function
/* Don't really give stack
                           thread2,
                          NULL,
                                                /* Size of stack needed
                          STACK_SIZE,
                          (PVOID) 0);
                                                /* Message to thread
    while (TRUE) {
         * Wait for a mouse event
         fWait= 1;
         MouReadEventQue(&mouEvent, &fWait, mouHand);
         if (mouEvent.time) {
             /*
 * First, check for a button 1 press:
             if (mouEvent.fs & MOUSE_BN1_DOWN) {
                      Wake the other thread up.
```

```
DosWriteQueue(
                                             /* Queue handle
                         buttonPressedQ,
                        0,
                                             /* User request ID
                                             /* Length of data passed
                        0.
                        NULL,
                                             /* Address of data buffer
                        0);
                                             /* Priority
                }
             * If button 2 is pressed, exit.
            if (mouEvent.fs & MOUSE_BN2_DOWN) {
                break;
        }
       Let's print something and go.
    printf("done.\n");
}
        thread2()
        Thread2 waits for request on the queue. When it gets one,
    it prints a message saying that a button was pressed.
void thread2(void *foo)
    REQUESTDATA
                        requestData;
    ULONG
                        dataLen:
    void
                        *dataBuf;
    BYTE
                        messagePrio:
     * Repeatedly wait for event, then process it.
    while (TRUE) {
        * Wait until a request appears
        dataLen= 0:
        DosReadQueue(
                                            /* Semaphore handle
                        buttonPressedQ,
                                             /* PID/request ID returned
                        &requestData,
                                            /* Length of data passed
/* Address of data passed
                        &dataLen,
                        &dataBuf,
                                            /* Remove first element
                        FALSE.
                                            /* Wait for a request
                        DCWW_WAIT,
                                            /* Message priority returned*/
                        &messagePrio.
                                             /* Semaphore (not required) */
                        NULLHANDLE);
            We've gotten a button press message. Let's process it.
        printf("Button 1 Pressed\n");
}
```

Deadlocks

Another danger when writing multithreaded programs is that of *deadlocks*. Sometimes called, rather romantically, *deadly embrace*, deadlocks cause a program to hang. The most simple deadlock is when thread A is waiting for thread B to do something and thread B is waiting for thread A to do something. Because both are waiting for the other, nothing happens; they are in a deadly embrace.

In program prog2-7.c, which is presented in Figure 2.7, we illustrate a common deadlock. Here, both threads are attempting to secure two resources. Because they do not request the resource in the same order, they each might get one of the two resources, then wait indefinitely for the other:

thread2 thread1 DosRequestMutexSem(semaphore1, SEM INDEFINITE WAIT); DosRequestMutexSem(semaphore2, SEM INDEFINITE WAIT); DosRequestMutexSem(semaphore2, SEM_INDEFINITE_WAIT); ... blocked ... DosRequestMutexSem(semaphore1, SEM INDEFINITE WAIT); ... blocked ...

After thread1 gets semaphore 1 and thread 2 gets semaphore 2, thread 1 cannot continue because it cannot get semaphore 2. Similarly, thread 2 cannot continue because it cannot get semaphore 1.

We will see in chapter 12 that avoiding deadlocks can be a subtle and difficult task.

Figure 2.7 prog2-7.c

```
/*

**

**

**

This program works shows a deadlock. Two threads compete for two resources. Because they request the resources in different orders, sometimes one thread will own one resource and the other thread will own the second resource. Both will be waiting for the other to relinquish its resource and nothing will happen.

**

#include <stdio.h>
#include <process.h>

#define INCL_DOSPROCESS
#define INCL_DOSSEMAPHORES
#include <os2.h>

#include "mt.h"
```

```
* Function declaration for the second thread
void thread2(void *);
 * Declare our resources
HMTX
                        resource1;
HMTX
                        resource2;
        Main()
void main()
{
    TID
                        thread2Tid;
    * Create our mutex semaphores:
    DosCreateMutexSem(
                                             /* We don't need a name
                        NULL,
                        &resource1,
                                             /* Place to return handle
                                             /* No options
                                            /* Initially unowned
                        FALSE):
   DosCreateMutexSem(
                                             /* We don't need a name
                        NULL,
                                            /* Place to return handle
/* No options
                        &resource2,
                                            /* Initially unowned
                        FALSE);
        Start the thread
    thread2Tid= _beginthread(
                        thread2,
                                            /* Address of function
                                            /* Don't really give stack
                        NULL,
                        STACK SIZE,
                                           /* Size of stack needed
                        (PVOID) 0):
                                            /* Message to thread
    while (TRUE) {
       /*

* Get resource 1
        DosRequestMutexSem(
                        resource1,
                        SEM INDEFINITE WAIT):
        printf("Thread 1 has resource 1\n");
        * Now get resource 2
        DosRequestMutexSem(
                        resource2,
                        SEM_INDEFINITE_WAIT);
        printf("Thread 1 has resource 2\n");
           Now release resource 2
        DosReleaseMutexSem(resource2);
```

Figure 2.7 Continued

```
printf("Thread 1 released resource 2\n");
        /*
 * And release resource 1
 */
        DosReleaseMutexSem(resource1);
        printf("Thread 1 release resource 1\n");
}
        thread2()
        Thread2 tries to obtain two resources. It does so in the
    reverse order from thread 1.
void thread2(void *foo)
{
   while (TRUE) {
        /*

* Get resource 2
        DosRequestMutexSem(
                        resource2,
                        SEM_INDEFINITE_WAIT);
        printf("Thread 2 has resource 2\n");
           Now get resource 1
        DosRequestMutexSem(
                        resource1,
                        SEM_INDEFINITE_WAIT);
        printf("Thread 2 has resource 1\n");
        /*

* Now release resource 1
        DosReleaseMutexSem(resource1);
        printf("Thread 2 released resource 1\n");
        /*

* And release resource 2
        DosReleaseMutexSem(resource2);
        printf("Thread 2 release resource 2\n");
```

Summary

Multithreading allows a single program to perform many tasks simultaneously. Writing multithreaded programs is difficult because timing and synchronization now becomes a factor in the execution of the program. Operating systems that provide multithreading also provide primitives for managing the timing of threads. Mutual exclusion primitives allow serialization of access to shared resources. Synchronization primitives allow threads to communicate timing information without the need to busy-wait. Queues allow convenient client/server relationships.

OS/2 Multithreading Facilities

Chapter

3

Starting and Ending Threads

This chapter explores how to begin and terminate the execution of a thread. When an OS/2 process begins execution, it has one thread running. To become "multi-threading," the program needs to create new threads.

There are two calls which are available to create threads: _beginthread() and DosCreateThread(). It might not be apparent immediately which is appropriate. In general, the C programmer always should use _beginthread(). This call goes through the C language runtimes, allowing them to create and initialize per-thread databases. If you create a thread with DosCreateThread(), the language runtimes will not have been allowed to initialize and calls from that thread to any runtimes (e.g. printf(), getchar()) might result in disaster.

The C language runtimes provide a very useful function called _threadstore(). This function allows each thread to easily get access to a unique storage area for that thread. Without this facility, the programmer would have two choices for data: global, which is shared by all threads in the process, and local, which is available only to the function in which it is defined or the functions to which it is passed as a parameter. The _threadstore() function allows for data that can be accessed easily by any function in a thread.

At the end of this chapter, the listings of four example programs are presented. These programs will help the reader to see exactly how the calls described in this chapter work. Peruse the programs, and use the function descriptions in the chapter as a reference to help you understand how they work.

Program prog3-1.c, which is presented in Figure 3.1, shows how simple it is to create threads. In it, the main thread creates two other threads, then waits for their completion. Each thread prints a message and beeps.

Figure 3.1 prog3-1.c

```
prog3-1.c
        This program demonstrates starting and ending threads.
    Two threads are started, each of which beeps and prints a
   message.
#include <stdio.h>
#include <process.h>
#define INCL_DOSPROCESS
#include <os2.h>
#include "mt.h"
   Function declarations for the threads.
void thread2(void *);
void thread3(void *);
        Main()
void main()
{
    TID
                        thread2Tid;
                        thread3Tid;
    printf("[Threads start...\n");
       Start the threads:
    thread2Tid= _beginthread(
                                            /* Address of function
                        thread2,
                                            /* Don't really give stack
                                            /* Size of stack needed
                        STACK_SIZE,
                                             /* Message to thread
                        (PVOID) 0);
    thread3Tid= _beginthread(
                                             /* Address of function
                        thread3,
                                             /* Don't really give stack
                        NULL,
                                             /* Size of stack needed
                        STACK_SIZE,
                                            /* Message to thread
                        (PVOID) 0);
        Wait for both tasks to complete.
    DosWaitThread(&thread2Tid, DCWW_WAIT);
    DosWaitThread(&thread3Tid, DCWW_WAIT);
        They're finished now.
    printf("end]\n");
```

```
thread2()
        Thread2 beeps the speaker and prints a message.
void thread2(void *foo)
    DosBeep (200, 300);
                          \"Hello World!\" said thread2.\n" ANSI_RESET);
    printf(ANSI_CYAN "
        When we return from this function, the thread
        will automatically terminate.
}
        thread3()
        Thread3 beeps the speaker and prints a message.
void thread3(void *foo)
    DosBeep (350,300);
    printf(ANSI_MAGENTA
                 \"Hello World!\" suggested thread3.\n"
             ANSI_RESET);
        Return, terminating the thread.
```

Program prog3-2.c, which is presented in Figure 3.2, is similar to prog3-1.c, except that, in this program, one thread kills another. Note that, because the threads are not synchronized in any way, one cannot be sure exactly where the thread will be killed. You probably will see that the killed task does not print out its message, having been killed while it was beeping.

Great care needs to be taken when using DosKillThread(). This function terminates a thread wherever it is in its execution. It could be holding resources when it is killed. For this reason, we recommend against using it except under the most controlled circumstances. It generally is better to use synchronization primitives to signal a thread to terminate itself rather than terminating it from a separate thread.

Program prog3-3.c, which is presented in Figure 3.3, shows how to use DosCreate Thread(). While we do not recommend using this call, we have shown how you would use it if you wanted to. Note that we have had to change our printf() calls to DosWrite() calls. Because we have used DosCreateThread() to start the thread, the language runtimes have not had an opportunity to initialize, so we cannot use runtime calls like printf(). In point of fact, we do cheat a little here, making a call to

strlen(), which is a C language runtime and as such should be off limits. However, it's a simple enough call, and we've stepped through the assembly language to verify that it doesn't use any per-thread data.

Figure 3.2 prog3-2.c

```
prog3-2.c
        This program demonstrates using DosKillThread() to kill
   another thread.
        Thread2 kills thread3, generally before thread3 gets to run
    to completion.
#include <stdio.h>
#include <process.h>
#define INCL DOSPROCESS
#include <os2.h>
#include "mt.h"
   Function declarations for the threads.
*/
void thread2(void *);
void thread3(void *);
   Make the TIDs of the threads globally available.
TID
                        thread2Tid;
TID
                        thread3Tid;
        Main()
void main()
    printf("Program starts:\n");
    printf("[Threads start...\n");
       Start the threads:
    thread2Tid= _beginthread(
                        thread2,
                                             /* Address of function
                                            /* Don't really give stack
                        NULL,
                                             /* Size of stack needed
                        STACK_SIZE,
                                             /* Message to thread
                        (PVOID) 0);
    thread3Tid= _beginthread(
                                             /* Address of function
                        thread3,
                        NULL,
                                             /* Don't really give stack
                        STACK SIZE.
                                             /* Size of stack needed
                                             /* Message to thread
                        (PVOID) 0);
        Wait for both tasks to complete.
```

```
DosWaitThread(&thread2Tid, DCWW_WAIT);
    DosWaitThread(&thread3Tid, DCWW_WAIT);
    printf("end]\n");
        thread2()
        Thread2 beeps, then kills thread3. Thread3 generally doesn't
    get a chance to run to completion.
void thread2(void *foo)
    DosBeep(200,300);
    DosKillThread(thread3Tid);
    printf(ANSI_RED " \"Hello World!\" said thread2.\n" ANSI_RESET);
        thread3()
        Thread3 beeps and prints a message.
void thread3(void *foo)
    DosBeep (350, 300);
    printf(ANSI_GREEN "
                           \"Hello World!\" suggested thread3.\n" ANSI_RESET);
Figure 3.3 prog3-3.c
        prog3-3.c
        This program demonstrates starting and ending threads.
    Rather than using _beginthread(), we use DosCreateThread(). As
    a result, the C language runtimes are not initialized, so we
   cannot use them. We therefore use DosWrite() to write our
   message, rather than printf().
#include <string.h>
#define INCL_DOSPROCESS
\#define\ INCL\_DOSFILEMGR
#define INCL_DOSEXCEPTIONS
#include <os2.h>
#include "mt.h"
```

Figure 3.3 Continued

```
* Define the standard channels.
#define HF_STDIN
                        0
#define HF STDOUT
^{\star} Function declarations for the threads.
FNTHREAD thread2;
FNTHREAD thread3:
       Main()
void main()
{
   TID
                        thread2Tid;
   TID
                        thread3Tid;
   ULONG
                       bytesWritten;
                       *p;
   char
   p= "[Threads start...\n\r";
   DosWrite(HF_STDOUT, p, strlen(p), &bytesWritten);
       Start the threads:
    */
   DosCreateThread(
                                          /* Returned thread ID
                        &thread2Tid,
                        thread2.
                                            /* Address of thread start
                        0,
                                           /* Message to pass
                                           /* No options
                        0,
                                           /* Size of stack
                        STACK_SIZE);
   DosCreateThread(
                       &thread3Tid,
                                            /* Returned thread ID
                        thread3,
                                           /* Address of thread start
                                           /* Message to pass
                        0,
                                           /* No options
                                           /* Size of stack
                        STACK SIZE);
       Wait for both tasks to complete.
   DosWaitThread(&thread2Tid, DCWW_WAIT);
   DosWaitThread(&thread3Tid, DCWW_WAIT);
       They're finished now.
   p= "end]\n\r";
   DosWrite(HF_STDOUT, p, strlen(p), &bytesWritten);
        thread2()
       Thread2 beeps the speaker and prints a message.
   Notice that the function type for threads created with
   DosCreateThread() is different from those created with
   beginthread().
```

```
void APIENTRY thread2(ULONG foo)
   ULONG
                        bytesWritten;
    char
    DosBeep (200,300);
    p= ANSI CYAN " \"Hello World!\" said thread2.\n\r" ANSI_RESET;
    DosWrite(HF_STDOUT, p, strlen(p), &bytesWritten);
        When we return from this function, the thread
        will automatically terminate.
}
        thread3()
        Thread3 beeps the speaker and prints a message.
void APIENTRY thread3(ULONG foo)
{
                        bytesWritten;
   UI ONG
    char
                       *p;
   DosBeep (350,300);
    p= ANSI_MAGENTA "
                         \"Hello World!\" suggested thread3.\n\r" ANSI_RESET;
   DosWrite(HF_STDOUT, p, strlen(p), &bytesWritten);
        Return, terminating the thread.
```

Program prog3-4.c, which is presented in Figure 3.4, finishes off the chapter by demonstrating the use of _threadstore(). The language runtimes reserve a pointer variable for each thread. The address of the pointer variable is returned by _threadstore(). In this program, we use this pointer variable to hold a pointer to a text string that we print out. A more standard use of this pointer variable is to hold a pointer to a per-thread data area to get easily accessible data for each thread.

Figure 3.4 prog3-4.c

Figure 3.4 Continued

```
#include <stdio.h>
#include <process.h>
#define INCL_DOSPROCESS
#include <os2.h>
#include "mt.h"
 ^{\star} Function declarations for the threads.
void thread2(void *);
void thread3(void *);
* Prototype for our printThreadstore() function
void printThreadstore(void);
        Main()
void main()
{
   TID
                        thread2Tid;
                        thread3Tid;
    TID
   printf("[Threads start...\n");
        Start the threads:
    thread2Tid= _beginthread(
                        thread2,
                                            /* Address of function
                                            /* Don't really give stack
                        NULL,
                                            /* Size of stack needed
                        STACK_SIZE,
                        (PVOID) 0);
                                            /* Message to thread
    thread3Tid= _beginthread(
                                             /* Address of function
                        thread3,
                                            /* Don't really give stack
                        NULL,
                        STACK_SIZE,
                                            /* Size of stack needed
                                            /* Message to thread
                        (PVOID) 0);
        Wait for both tasks to complete.
    DosWaitThread(&thread2Tid, DCWW_WAIT);
    DosWaitThread(&thread3Tid, DCWW_WAIT);
     * They're finished now.
    printf("end]\n");
}
        thread2()
        Thread2 beeps the speaker and prints a message.
```

```
void thread2(void *foo)
    *_threadstore()= ANSI_CYAN
                          \"Hello World!\" said thread2.\n"
                     ANSI_RESET;
   printThreadstore();
}
        thread3()
        Thread3 beeps the speaker and prints a message.
void thread3(void *foo)
    *_threadstore()= ANSI_MAGENTA
                        \"Hello World!\" suggested thread3.\n"
                  ANSI_RESET;
   printThreadstore();
}
       printThreadstore()
       Assumes that the location returned by _threadstore() points
    to a string. Gets the string and prints it.
void printThreadstore()
   printf((char*) *_threadstore());
}
```

Summary

There are two ways to begin auxiliary threads: DosCreateThread() and _begin thread(). We suggest using _beginthread() because it allows the thread to use the C runtime libraries. A thread can end in one of three ways: it can return from its main function, it can issue an _endthread() call, or it can be the target of a DosKillThread() call.

_beginthread()

The _beginthread() function is used to create a new thread in the current process.

Prototype

Parameters

The address of the function that the new thread will execute.

stack Pointer to a stack for the thread. This parameter is ignored in OS/2

2.x but is retained for compatibility.

stackSize The size of the stack that _beginthread() will create. Stack size

should be a multiple of 4096 bytes and be at least 8192 bytes.

message A user-defined message that will be passed as the sole parameter to

the thread's main function.

Return value

On successful creation of a thread, _beginthread() returns the thread id of the new thread. If an error occurs a -1 is returned.

Description

To start a thread, one can call either _beginthread() or DosCreateThread(). It generally is better to use _beginthread(), because this call gives the C language runtimes an opportunity to initialize itself. If a thread is created with DosCreate Thread(), that thread must not call any C language runtimes, such as printf() or getchar().

The created thread begins execution at the beginning of the specified function. When the function returns, the thread terminates.

_endthread()

The _endthread() function terminates a thread.

Prototype

```
#include     void _endthread(void);
```

Description

The <code>_endthread()</code> function permits a thread to explicitly terminate itself. A thread can terminate itself either by returning from the main thread function or by issuing an <code>_endthread()</code> anywhere in the execution of the thread.

_threadstore()

The _threadstore() function provides access to per-thread data areas.

Prototype

```
#include     void **_threadstore(void);
```

Return value

A pointer to a location large enough to hold a (void *) pointer is returned. This location is unique to each task.

Description

This function allows the programmer to have global per-thread data. Normally, static variables can be used for global data. When multithreading, however, static variables are shared between threads. Using _threadstore(), each task can have a unique data area for its own use.

DosCreateThread()

The DosCreateThread() function is used to create a new thread in the current process.

Prototype

Parameters

ptid A pointer to the location where the TID of the new thread should be returned.

pfn The address of the function that the new thread will execute.

param A user-defined message that will be passed as the sole parameter to the

thread's main function.

flag If bit 0 is set, the new thread will begin in a suspended state, and a

 ${\tt DosResumeThread}$ () will be required to start it running. If bit 0 is not

set, then the thread will start ready-to-run.

cbStack The size of the stack that DosCreateThread() will create. Stack size

should be multiple of 4096 bytes and be at least 8192 bytes.

Return value

- 0 NO ERROR
- 8 ERROR_NOT_ENOUGH_MEMORY
- 95 ERROR INTERRUPT
- 115 ERROR PROTECTION VIOLATION
- 164 ERROR_MAX_THREADS_REACHED

Description

To start a thread, one can call either _beginthread() or DosCreateThread(). It generally is better to use _beginthread(), because this call gives the C language runtimes an opportunity to create per-thread databases. If a thread is created with DosCreateThread(), that thread must not call any C language runtimes, such as printf() or getchar().

46 OS/2 Multithreading Facilities

One feature of DosCreateThread() that is not supported by _beginthread() is the ability to start a thread in a suspended state. This feature, however, can be simulated easily with an event semaphore.

The created thread begins execution at the beginning of the specified function. When the function returns, the thread terminates.

DosKillThread()

The DosKillThread() function terminates a thread in the same process.

Prototype

Parameter

tid Holds the thread ID of the thread to terminate.

Return value

```
0 NO_ERROR
170 ERROR_BUSY
309 ERROR_INVALID_THREADID
```

Description

The DosKillThread() function permits one thread to terminate the execution of another thread in the process. It returns to the caller without waiting for the thread to be terminated. You should not use the DosKillThread() function to terminate the currently executing thread. You should use the _endthread() function for that purpose.

DosWaitThread()

The DosWaitThread() function allows a thread to wait for the termination of another thread.

Prototype

Parameters

Pointer to thread ID of thread to wait for, or 0 to wait for any thread. If *ptid is zero on entry, it will contain the thread ID of the thread that terminated on exit.

option DCWW_WAIT to wait for thread to terminate, or DCWW_NOWAIT to return immediately.

Return value

```
0 NO_ERROR
294 ERROR_THREAD_NOT_TERMINATED
309 ERROR_INVALID_THREADID
```

Description

This function allows a thread to wait for or check for completion of another thread. It has two basic modes of operation. It can wait for a specific thread, by specifying the thread's ID, or it can wait for any thread termination in the process, by specifying 0 for the thread ID.

Chapter

4

Enabling and Disabling Thread Rescheduling

This chapter presents the system calls used to enable and disable task rescheduling. To ensure correct execution of programs, it is sometimes necessary to prevent other threads in the same process from running. This can be achieved by issuing the call DosEnterCritSec(). This call tells the operating system that we do not want any other thread in our process to be allowed to run until we issue a DosExitCritSec().

Why would we want to prevent rescheduling? Program prog2-1.c (see Figure 2.1) gives an example of a race condition that is corrected by using DosEnterCritSec() and DosExitCritSec() in prog2-2.c (see Figure 2.2).

Care needs to be taken in using these calls. In between the calls, no other thread will be allowed to run. If some other thread is holding a resource and the current thread attempts to secure that resource, a deadlock will result. The current thread will block waiting for another thread to release a resource, but the other thread will not be allowed to run, so it will be unable to release the resource. As a rule of thumb, do not put code into a critical section that might block. This includes file I/O, screen I/O, and semaphore waits.

Program prog4-1.c, which is presented in Figure 4.1, illustrates how critical sections disable scheduling. It is not a realistic example of how critical sections are used in practice, but it allows you to see when scheduling is enabled and disabled.

Summary

Scheduling of other threads in the same process can be disabled by calling Dos EnterCritSec() and reenabled by calling DosExitCritSec(). This allows sensitive sequences of code to be protected from being interrupted by other code in the same process.

Figure 4.1 prog4-1.c

```
prog4-1.c
        This program demonstrates how DosEnterCritSec() disables
   thread rescheduling.
        Thread2 repeatedly writes a message. Thread1 disables
   scheduling, sleeps for a while, and re-enables it. During the
   time that scheduling is disabled, thread2 cannot run.
#include <stdio.h>
#include <process.h>
#define INCL_DOSPROCESS
#define INCL_DOSSEMAPHORES
#include <os2.h>
#include "mt.h"
   Function declarations for the threads.
void thread2(void *);
        Main()
void main(int argc, char *argv[])
                        thread2Tid:
   ULONG
        Start the thread:
    thread2Tid= _beginthread(
                        thread2.
                                             /* Address of function
                                             /* Don't really give stack
                        STACK_SIZE,
                                            /* Size of stack needed
                                            /* Message to thread
                        (PVOID) 0);
    for (i = 0; i < 3; i++) {
        /*

* Let the other thread run a while.
        DosSleep(3000);
            Now disable rescheduling.
        DosEnterCritSec();
         ^{\ast} \, No other threads in the program can run now. Sleep for
           awhile, effectively suspending the whole program.
```

```
DosSleep(5000);
             ^{\prime*} ^{\ast} OK, re-enable scheduling, allowing the other thread to run.
            DosExitCritSec();
}
             thread2()
             Thread2 prints a message over and over again.
void thread2(void *thread2um)
{
                                      wheel[]= "-\\!/";
      char
      int
                                      i;
      i = 0;
      for (;;) {
            /*

* We make the printf() a critical section. We do not want

* Standard to disable rescheduling while we are
            * We make the printi() a critical section. We do not want the other thread to disable rescheduling while we are in the middle of the write. We (the system on our behalf) might be holding global resources. This would cause the presentation manager to hang while the first thread is sleeping.
            DosEnterCritSec();
            printf("Thread 2 running " ANSI_CYAN "%c" ANSI_RESET "\n",
                                wheel [(i++ / 3) \% 4]);
             ^{\prime*} ^{} OK, we can re-enable scheduling now.
            DosExitCritSec();
}
```

DosEnterCritSec()

The DosEnterCritSec() function disables all thread rescheduling for the current process.

Prototype

```
#define INCL_DOSPROCESS
#include <os2.h>
APIRET DosEnterCritSec(void);
```

Return value

- 0 NO ERROR
- 30 ERROR INVALID THREADID
- 484 ERROR CRITSEC OVERFLOW

Description

The DosEnterCritSec() function disables all thread switching for the current process. The current thread will be the only thread in the current process to run until a DosExitCritSec() is issued. Note that, once a DosEnterCritSec() function is called, under no circumstances should dynamic link function calls be made until DosExitCritSec() is called. Doing so could cause a deadlock if the thread requests a resource which is held by another thread.

Calls to DosEnterCritSec() can be nested. Thus, if three calls to DosEnterCritSec() are issued, three calls to DosExitCritSec() must be issued to reenable scheduling.

There is one exception to the exclusivity of critical sections. If a signal occurs, thread 1 will be allowed to execute its exception handling code even if another thread in the same process is in a critical section. For this reason, exception handling code must be carefully written to avoid performing any operations that might interfere with a thread in a critical section.

DosExitCritSec()

The DosExitCritSec() function reenables thread scheduling following a Dos EnterCritSec().

Prototype

```
#define INCL_DOSPROCESS
#include <os2.h>
APIRET DosExitCritSec(void);
```

Return value

```
0 NO_ERROR
309 ERROR_INVALID_THREADID
484 ERROR_CRITSEC_OVERFLOW
```

Description

The DosExitCritSec() function reenables scheduling of other threads in the current process. If multiple DosEnterCritSec() calls have been issued, then the same number of DosExitCritSec() calls are required before scheduling actually will be enabled.

Chapter

5

Suspending and Resuming Thread Execution

In the previous chapter, you saw how to prevent all other threads in the current process from executing. It sometimes is convenient to be able to stop a particular thread from executing. The functions <code>DosSuspendThread()</code> and <code>DosResume Thread()</code> provide this capability. We do not advocate their use for the following reason: they are not sufficiently controllable to prevent deadlocks.

A call to DosSuspendThread() stops a target thread dead in its tracks. That thread might be holding resources (see chapter 7). If the current thread attempts to secure resources held by the suspended thread, a deadlock could result.

We give a simple example of using DosSuspendThread() and DosResume-Thread() in prog5-1.c, which is presented in Figure 5.1. In this program, we have a thread that repeatedly writes to the screen. The main thread waits, suspends the target thread, waits some more, and resumes the target thread. If you run this program in a VIO window, you will most likely find that the Presentation Manager is hung for the duration of the suspend. This is because the thread that is the target of the DosSuspendThread(), the writing thread, is writing to the screen and has the whole screen locked for the duration of the write. It is suspended while owning the screen, which prevents other threads in the system from being able to access the screen.

We have solved this problem in prog5-2.c, which is presented in Figure 5.2, by using a mutual exclusion semaphore around the DosSuspendThread() call and around the screen write. This way, we know that, when we suspend the thread, it is not, at that moment, writing to the screen. Thus, the Presentation Manager does not hang.

Rather than using DosSuspendThread() and DosResumeThread(), we suggest using more controlled techniques such as event semaphores. In chapters 7 and 8, several examples are presented that demonstrate the use of event semaphores to block threads. These approaches are better than DosSuspendThread() in that you know where a thread is when it blocks, thus preventing many deadlocks.

Figure 5.1 prog5-1.c

```
prog5-1.c
       This program demonstrates how DosSuspendThread() and
   DosResumeThread() work.
       Thread 2 repeatedly writes to the screen. Thread 1 waits
   awhile, then suspends thread 2. It waits some more and resumes
    thread 2.
   WARNING!!!! This program might cause the presentation manager
   to hang while the second thread is suspended. This happens
   because the thread is writing to the screen and has the screen
    locked when it is suspended.
#include <stdio.h>
#include <process.h>
#define INCL_DOSPROCESS
#include <os2.h>
#include "mt.h"
   Function declaration for the thread.
void thread2(void *);
       Main()
void main()
    TID
                        thread2Tid;
       Start the thread
    thread2Tid= _beginthread(
                        thread2.
                                            /* Address of function
                                            /* Don't really give stack
                        NULL.
                                            /* Size of stack needed
                        STACK SIZE,
                        (PVOID) 0);
                                            /* Message to thread
    * Hang out a little while:
    DosSleep(5000);
       Now suspend the thread
    DosSuspendThread(thread2Tid);
    /*

* Let's wait a while
```

```
DosSleep(5000);
     * OK, let's wake it back up now.
    DosResumeThread(thread2Tid);
       Wait a little while longer.
    DosSleep(5000);
       We're all done now.
    printf("done.\n");
        thread2()
        Thread2 repeatedly gets the time of day and prints it.
void thread2(void *foo)
    DATETIME
                        dateTime;
    while (TRUE) {
       DosGetDateTime(&dateTime);
       printf("Thread 2 running at %2d:%02d:%02d\n",
                    dateTime.hours.
                    dateTime.minutes,
                    dateTime.seconds);
}
```

Figure 5.2 prog5-2.c

#include <stdio.h>

Figure 5.2 Continued

```
#include <process.h>
#define INCL_DOSSEMAPHORES
#define INCL_DOSPROCESS
#include <os2.h>
#include "mt.h"
void thread2(void *);
    Declare a mutual exclusion semaphore to solve
   the hanging problem.
HMTX
                          systemLock;
        Main()
void main()
    TID
                          thread2Tid;
        Create the lock:
    DosCreateMutexSem(
                                               /* We don't need a name
/* Place to return handle
/* No options
/* Initially unowned
                          &systemLock,
        Start the thread
    thread2Tid= _beginthread(
                                                /* Address of function
                          thread2,
                                                /* Don't really give stack
/* Size of stack needed
                          NULL,
                          STACK SIZE.
                                                /* Message to thread
                          (PVOID) 0);
     * Hang out a little while:
*/
    DosSleep(5000);
    /*
 * Secure systemLock semaphore
 */
    DosRequestMutexSem(systemLock, SEM_INDEFINITE_WAIT);
        Now suspend the thread
```

```
DosSuspendThread(thread2Tid);
    '* We can release the semaphore now
   DosReleaseMutexSem(systemLock);
    * Let's wait a while
*/
   DosSleep(5000);
   DosResumeThread(thread2Tid);
    /*
* Wait a little while longer.
   DosSleep(5000);
    * We're all done now.
   printf("done.\n");
       thread2()
       Thread2 repeatedly gets the time of day and prints it.
void thread2(void *foo)
   DATETIME
                       dateTime;
   while (TRUE) {
       DosGetDateTime(&dateTime);
       /*
* Secure systemLock semaphore
       DosRequestMutexSem(systemLock, SEM_INDEFINITE_WAIT);
       printf("Thread 2 running at %2d:%02d:%02d\n",
                   dateTime.hours,
                   dateTime.minutes,
                   dateTime.seconds);
           We can release it now.
       DosReleaseMutexSem(systemLock);
}
```

60 OS/2 Multithreading Facilities

Summary

One thread can suspend another thread by issuing a DosSuspendThread() call. Great care must be taken when using this technique, as the thread doing the suspending might not know where in its execution the target thread is. Thus, the target thread might be suspended while holding resources.

DosSuspendThread()

The DosSuspendThread() is used by one thread to temporarily suspend the execution of another thread in the same process.

Prototype

```
#define INCL_DOSPROCESS
#include <os2.h>
APIRET DosSuspendThread(TID tid);
```

Parameter

tid The thread ID of the thread that is to be suspended.

Return value

```
0 NO_ERROR
309 ERROR INVALID_THREADID
```

Description

The DosSuspendThread() function temporarily suspends the execution of a target thread until the DosResumeThread() function is called. Under certain circumstances, the thread might not stop executing right away because some system resources might need to be freed up. The thread, however, will not execute any further thread instructions until the DosResumeThread() function is called.

DosResumeThread()

The DosResumeThread() function resumes the execution of a thread that previously has been suspended by calling DosSuspendThread().

Prototype

```
#define INCL_DOSPROCESS
#include <os2.h>
APIRET DosResumeThread(TID tid);
```

Parameter

tid The thread ID of the thread that is to be resumed.

Return value

- 0 NO_ERROR
- 90 ERROR NOT FROZEN
- 309 ERROR_INVALID_THREADID

Description

The DosResumeThread() function is used to resume the execution of a previously suspended thread. If the thread whose execution is to be resumed is not in a suspended state, then the ERROR_NOT_FROZEN code is returned.

Chapter

6

Changing the Priority of a Thread

It sometimes is useful to assign different priorities to threads so that some threads will receive more CPU time or better response time. OS/2 provides prioritized scheduling with round-robin scheduling within a priority. A thread priority consists of a priority class and a priority level. There are four priority classes: time critical, server, regular, and idle. The general priority class is regular.

The highest priority class is time critical. A thread should be assigned to the time critical priority class if it needs to be scheduled quickly. Suppose, for example, that you have a special hardware device and that the driver for that device has a limited buffer space. The thread reading the data should be made time critical to ensure that no data is lost as a result of other threads monopolizing the CPU. Time critical threads are guaranteed to be scheduled within 6ms of becoming ready-to-run. The next priority class is server.

Server threads are not time critical but need to be scheduled before regular priority threads. They are intended for server processes in client/server situations. They are more critical than standard user program threads, but they are not time critical.

Most program threads will have the regular priority class. OS/2 provides several tweaks in the scheduler to ensure that regular priority threads are scheduled in a timely manner. Recent behavior of a thread as well as whether or not it is part of a foreground process (a process that has the input focus) is taken into account. Unless there is a compelling reason to do otherwise, program threads should be kept in the regular priority class.

The lowest priority class is idle. Threads in this class are scheduled when there literally is nothing else to do.

Within a priority class, threads are assigned a priority level. Priority levels are numbers from 0 to 31. The higher the number the higher the priority for scheduling.

When two or more threads of the same priority class and level are ready to run, the one that was run least recently is run. Thus, they are scheduled in a round-robin fashion.

Program prog6-1.c, which is presented in Figure 6.1, demonstrates the use of <code>DosSetPriority()</code>. The program takes two arguments: the priority levels for two threads. It starts two threads with the two priority levels. The two threads each increment their own counter. After a delay, the threads are terminated and the counters are examined. We can see the relative amount of CPU time that the two threads received by noting the relative values of their counters. It is instructive to try the program with various priority levels.

Figure 6.1 prog6-1.c

```
prog6-1.c
        This program demonstrates DosSetPriority to set the
    priority of a thread.
        This program takes two arguments: the priorities that the
    two threads are to run. Each thread increments a different
    global counter. After a while, the main thread kills the other
    two threads and sees how high the counts got. This gives some
    idea of the relative amount of CPU time that the two threads
    received.
#include <stdio.h>
#include <stdlib.h>
#include <process.h>
#define INCL_DOSPROCESS
#include <os2.h>
#include "mt.h"
   Function declarations for the threads.
void thread2(void *);
void thread3(void *);
   Declare our global progress counters
*/
long
                        counter1:
long
                        counter2:
        Main()
void main(int argc, char *argv[])
    TID
                        thread2Tid:
    TID
                        thread3Tid;
    int
                        thread2Prty;
    int
                        thread3Prty;
    ULONG
                        err;
    if (argc != 3) {
```

```
printf("Usage: %s <thread2 priority> <thread3 priority>\n",
                arqv[0]);
    return:
    }
thread2Prty= atoi(argv[1]);
thread3Prty= atoi(argv[2]);
printf("Threads will have the following priorities:\n");
printf(" thread2: %d\n", thread2Prty);
printf("
          thread3: %d\n", thread3Prty);
counter1= 0;
counter2= 0:
* Start the threads, set the priorities, but don't allow them to run.
DosEnterCritSec();
  First, make sure the main task (me) gets CPU whenever it wants
DosSetPriority(
                   PRTYS THREAD,
                                      /* Scope is just the thread */
                   PRTYC_TIMECRITICAL, /* Time critical scheduling */
                                   /* Priority O
                   0.
                   1):
                                       /* Thread 1 (main thread)
* Start the threads:
thread2Tid= _beginthread(
                                      /* Address of function
                    thread2,
                   NULL.
                                      /* Don't really give stack */
                                      /* Size of stack needed
                   STACK SIZE.
                                      /* Message to thread
                   (PVOID) 0);
thread3Tid= _beginthread(
                   thread3,
                                       /* Address of function
                   NULL.
                                      /* Don't really give stack */
                   STACK_SIZE,
                                       /* Size of stack needed
                   (PVOID) 0):
                                       /* Message to thread
   We get the requested priority by first using a delta of -31 to
* ensure that the task priority is O, then using a delta of
* threadNPrty to increment the O priority to the requested priority.
if (err= DosSetPriority(
                                       /* Scope is just thread
                   PRTYS THREAD,
                                       /* Standard scheduling class*/
                   PRTYC_REGULAR,
                   -31.
                                       /* Ensure that priority is O*/
                                       /* Thread ID
                   thread2Tid)) {
   printf("Error: DosSetPriority, thread2, #%8d\n", err);
   return;
   }
if (err= DosSetPriority(
                                       /* Scope is just thread
                   PRTYS_THREAD,
                   PRTYC_REGULAR,
                                       /* Standard scheduling class*/
                                       /* Ensure that priority is O*/
                   -31.
                                      /* Thread ID
                   thread3Tid)) {
   printf("Error: DosSetPriority, thread3, #%8d\n", err);
```

}

Figure 6.1 Continued

```
return;
       }
   if (err= DosSetPriority(
                                            /* Scope is just thread
                        PRTYS_THREAD,
                                            /* Standard scheduling class*/
                       PRTYC_REGULAR,
                        thread2Prty,
                                            /* Requested priority
                                           /* Thread ID
                        thread2Tid)) {
       printf("Error: DosSetPriority, thread2, #%8d\n", err);
       return;
   if (err= DosSetPriority(
                        PRTYS_THREAD,
                                            /* Scope is just thread
                                            /* Standard scheduling class*/
                       PRTYC_REGULAR,
                                            /* Requested priority
                        thread3Prty,
                                           /* Thread ID
                        thread3Tid)) {
       printf("Error: DosSetPriority, thread3, #%8d\n", err);
       return;
       }
       OK, they're all set. Allow them to run.
   DosExitCritSec():
      Let them run for ten seconds and kill them.
   DosSleep(10000);
   DosKillThread(thread2Tid);
   DosKillThread(thread3Tid);
       Now see how they ran:
   printf("Task 1 got to %10d - %5.21f%%\n",
                counter1,
                (double)counter1 / (double)(counter1+counter2) * 100.0);
   printf("Task 2 got to %10d - %5.21f%%\n",
                counter2,
                (double)counter2 / (double)(counter1+counter2) * 100.0);
        thread2()
        Thread2 increments the global location "counter1" until in
void thread2(void *foo)
    for (;;) {
        counter1++;
```

```
/*
    thread3()

**

Thread3 increments the global location "counter2" until it
    is killed.

*

void thread3(void *foo)

{
    for (;;) {
        counter2++;
    }
}
```

Program prog6-2.c, which is presented in Figure 6.2, shows how a thread can find out what priority it is running at. It makes use of the DosGetInfoBlocks() call, which returns blocks containing information about the current process and the current thread.

Figure 6.2 prog6-2.c

```
prog6-2.c
        This program demonstrates DosSetPriority to set the
    priority of a thread.
        This program demonstrates how to determine the priority
    class and level of the current thread. It uses the
    DosGetInfoBlocks() call to get the thread information block.
#include <stdio.h>
#include <stdlib.h>
#include <process.h>
#define INCL DOSPROCESS
#include <os2.h>
#include "mt.h"
  Function declarations
void printPrio(void);
       Main()
void main(int argc, char *argv[])
    int
                        i;
       Do some loops to diddle the priority
```

Figure 6.2 Continued

```
for (i = 0: i < 40: i++) {
        DosSetPriority(
                         PRTYS THREAD.
                                               /* Scope is just the thread */
                                               /* Time critical scheduling */
                         PRTYC_NOCHANGE,
                         2,
                                               /* Up it by 2
                                              /* Thread 1 (main thread)
                         1);
        printPrio();
    for (i = 0; i < 16; i++) {
        DosSetPriority(
                                              /* Scope is just the three //
/* Time critical scheduling */
*/
                         PRTYS THREAD.
                         PRTYC NOCHANGE,
                         -1.
                                               /* Thread 1 (main thread)
                         1);
        printPrio();
    DosSetPriority(
                         PRTYS THREAD,
                                               /* Scope is just the thread */
                                               /* Time critical scheduling */
                         PRTYC_NOCHANGE,
                                               /* Make it priority 0
                                               /* Thread 1 (main thread)
                         1);
    printPrio();
}
void printPrio()
    PTIB
                         threadInfo;
    PPIB
                         processInfo:
        Get the info blocks.
    DosGetInfoBlocks(&threadInfo, &processInfo);
        Now print out the information. Note that the class and
        priority are encoded bytewise in the tib2_ulpri field.
    printf("Thread %d: Priority class %d, priority level %d\n",
             threadInfo->tib_ptib2->tib2_ultid,
             (unsigned)threadInfo->tib_ptib2->tib2_ulpri >> 8,
             threadInfo->tib_ptib2->tib2_ulpri & OxFF);
}
```

Summary

Threads have priority classes and priority levels. Thread priority is used by the scheduler to choose which thread to run. In short, the scheduler chooses the highest priority thread that is ready-to-run.

DosSetPriority()

The DosSetPriority() function alters the priority class and level of a thread executing in the current process.

Prototype

Parameters

scope Holds the scope of the priority change. The values are:

PRTYC_PROCESS All threads in the process

PRTYC_PROCESSTREE All threads in the process and its descendents

PRTYC_THREAD A single thread in the current process

class Holds the priority class of a process. The values are:

PRTYC_NOCHANGE Leave priority class unchanged

PRTYC_IDLETIME Set class to idle
PRTYC REGULAR Set class to regular

PRTYC_FOREGROUNDSERVER Set class to server PRTYC_TIMECRITICAL Set class to time critical

delta Holds the number to be added to the current priority of the target.

delta must be between -31 and +31. -31 guarantees that the new level

will be 0, +31 guarantees that the new level will be 31.

PorTid Holds a thread ID if scope is PRTYC_THREAD or a process ID otherwise.

Return value

```
0 NO_ERROR
303 ERROR_INVALID_PROCID
304 ERROR_INVALID_PDELTA
305 ERROR_NOT_DESCENDENT
307 ERROR_INVALID_PCLASS
308 ERROR_INVALID_SCOPE
309 ERROR_INVALID_THREADID
```

Description

The DosSetPriority() function changes the priority of a thread executing in the current process, all the threads of the current process, or all of the threads of the current process and all of its descendents. The programmer is permitted to alter the priority level by an amount ranging from -31 to +31. Negative numbers decrease the priority level of the target threads, and positive numbers increase the priority level. Within a given priority class, threads with high priorities are given preference over threads with lower priorities. Threads assigned equal priorities will operate in a round-robin fashion. From highest to lowest, priorities are as follows:

PRTYC_TIMECRITICAL	31
PRTYC_TIMECRITICAL	30
•	
:	
PRTYC_TIMECRITICAL	1
PRTYC_TIMECRITICAL	0
PRTYC_FOREGROUNDSERVER	31
PRTYC_FOREGROUNDSERVER	30
•	
•	
PRTYC_FOREGROUNDSERVER	1
PRTYC_FOREGROUNDSERVER	0
PRTYC_REGULAR	31
PRTYC_REGULAR	30
•	
•	
PRTYC_REGULAR	1
PRTYC_REGULAR	0
PRTYC_IDLETIME	31
PRTYC_IDLETIME	30
•	
•	
PRTYC_IDLETIME	1
PRTYC_IDLETIME	0

Chapter

7

Mutual Exclusion Semaphores

In chapter 2, we showed a basic problem of concurrency: mediating access to shared resources. A resource can be something as simple as a global variable, or it can be a more complex object like a file or a window. The purpose of mutual exclusion semaphores is to allow locking of shared resources so that only one thread at a time has access to them. The idea is this: only one thread at a time can "own" a mutex semaphore. Thus, if I associate a semaphore with a global resource and I access the resource only when I own the semaphore, then only one thread at a time will be accessing the resource. This requires cooperation: if code exists that accesses the resource without owning the semaphore, then no code can be assured of exclusive access. There is no explicit association between the semaphore and the resource that it is protecting. There merely is the programmer's understanding that the semaphore must be owned before the resource can be accessed.

Ownership of a semaphore is obtained by calling <code>DosRequestMutexSem()</code>. If the semaphore is not owned currently, the call will return immediately and the current thread will be the owner of the semaphore. If an attempt is made by another thread to get ownership of the semaphore, that thread will block until it can be granted ownership. A thread relinquishes ownership by calling <code>DosReleaseMutexSem()</code>. If another thread is waiting for ownership of the semaphore, then ownership is passed to that thread. Threads are granted ownership in a first-come first-served fashion. Thus, when a semaphore is released, the thread that has been waiting the longest is granted ownership.

Calls to DosRequestMutexSem() can be nested. If the owner of the semaphore calls DosRequestMutexSem(), the call returns immediately and the "request count" is incremented. The request count represents the number of times that the owner must call DosReleaseMutexSem() to relinquish ownership of the semaphore. This is an important property.

Suppose you have a routine A that accesses global resource X. Routine A needs to get ownership of the semaphore associated with global resource X to ensure orderly access, because it might be called from a context where X is not owned. However, what if routine B accesses X and calls routine A in the middle of its accesses to X. B, too, needs to get ownership before accessing X. Thus, the thread will own the semaphore when A is called, and we cannot have a second call to DosRequestMutex Sem() block as this would hang the program. Similarly, when A calls DosRelease MutexSem(), the thread still must own the resource, because B requested it. Thus, the resource is not freed until the second call to DosReleaseMutexSem().

Shared semaphores can be accessed by processes other than the creator of the semaphore. For other processes to access a semaphore, they must use DosOpen MutexSem() supplying either the name of the semaphore (for a named semaphore) or the semaphore handle (for an unnamed shared semaphore).

Program prog7-1.c, which is presented in Figure 7.1, shows how having two threads accessing a global resources can cause problems. Thread 2 reads the two values, value1 and value2 from the structure resource. Thread 3 is constantly incrementing the two values. Thus, the two values should stay in step and always be equal. Running the program, however, we find that they are unequal often enough. The problem comes if a reschedule occurs after thread 2 has read the first value but before it has read the second value. When thread 2 is scheduled again, it will find the second value to be greater than the first. The other potential race condition is if a reschedule occurs after thread 3 has incremented the first value but before it has incremented the second. In this can, thread 2 will find that value2 is 1 greater than value1. This is a simple example, but it shows that we cannot even maintain a simple constraint on our data structure. We cannot ensure that two values are equal.

Figure 7.1 prog7-1.c

```
prog7-1.c
        This program demonstrates the need for mutual exclusion
   semaphores.
        The main thread creates two threads. Thread 3 repeatedly
   modifies a global structure. Thread 2 reads the global structure,
   looks busy for a while, then reads again. If the two reads give
   different values, then the structure is inconsistent.
        We will fix this in the next program using a mutual
    exclusion semaphore.
#include <stdio.h>
#include <stdlib.h>
#include <process.h>
#define INCL_DOSPROCESS
#define INCL_DOSSEMAPHORES
#include <os2.h>
#include "mt.h"
```

```
Function declarations for the threads.
void thread2(void *);
void thread3(void *);
   Delcare the global "resource"
struct {
                          value1;
    long
                          value2;
    long
                          resource;
        Main()
void main(int argc, char *argv[])
                          thread2Tid;
    TID
    TID
                          thread3Tid;
        Start the threads:
    thread2Tid= _beginthread(
                                                /* Address of function
                          thread2,
                                                /* Don't really give stack
                          NULL,
                                                /* Size of stack needed
                          STACK_SIZE,
                                                /* Message to thread
                          (PVOID) 0);
    thread3Tid= _beginthread(
                                               /* Address of function
/* Don't really give stack
/* Size of stack needed
/* Message to thread
                          thread3.
                          NULL,
                          STACK_SIZE,
                          (PVOID) 0);
        Wait for thread 2 to finish
    DosWaitThread(&thread2Tid, DCWW_WAIT);
       Thread 2's done, kill thread 3
    DosKillThread(thread3Tid);
}
         thread2()
        Thread2 simulates a thread working with a global resource.
    It reads a global structure, then pretends to work with it for
    awhile and reads it again. If the two reads produce different
    results, it complains.
void thread2(void *foo)
```

Figure 7.1 Continued long localValue1; long long localValue2; long Let the other thread run a bit. DosSleep(1000); /* * Now pretend to do useful work for (i = 0; i < 100000; i++) { /*
* Read the global resource localValue1= resource.value1; /* * Act busy for $(tmp= 0; tmp < 10; tmp++) {$ /* * Read the resource again. localValue2= resource.value2; * Now see if the two reads produced consistent results if (localValue1 != localValue2) { printf("The resource was corrupted\n" " localValue1= %d, localValue2= %d\n", localValue1, localValue2); return; A miracle ! We made it ! printf("Thread ran to completion successfully\n"); } thread3() Thread3 increments the values in the structure "resource" until it is killed.

```
void thread3(void *foo)
{
    for (;;) {
```

```
resource.value1++;
resource.value2++;
}
}
```

Program prog7-2.c, which is presented in Figure 7.2, corrects this problem using a mutex semaphore. Both threads request the semaphore before accessing the resource and release the semaphore when they have finished accessing the resource. It might be instructive to remove the DosRequestMutexSem() and DosRelease MutexSem() in one of the two threads. You will find that the program behaves much as prog7-1 did. This illustrates an important point: mutex semaphores are useless unless they are used consistently everywhere that a resource is accessed.

Figure 7.2 prog7-2.c

```
prog7-2.c
        This program demonstrates how mutual exclusion is used to
    solve the problem of prog7-1.
        The main thread creates two threads. Thread 3 repeatedly
   modifies a global structure. Thread 2 reads the global structure,
    looks busy for a while, then reads again. If the two reads give
    different values, then the structure is inconsistent.
        Unlike the previous program, we use a mutex semaphore to
    ensure that only one thread accesses the resource at a time.
    This ensures that both threads will always see consistent
    results.
#include <stdio.h>
#include <stdlib.h>
#include <process.h>
#define INCL DOSPROCESS
#define INCL_DOSSEMAPHORES
#include <os2.h>
#include "mt.h"
   Function declarations for the threads.
void thread2(void *):
void thread3(void *);
   Delcare the global "resource"
struct {
   long
                        value1;
   long
                        value2;
                        resource;
```

Figure 7.2 Continued

```
* Declare our Mutual Exclusion semaphore
HMTX
                        lock;
        Main()
void main(int argc, char *argv[])
    TID
                         thread2Tid;
    TID
                         thread3Tid;
        Create the mutual exclusion semaphore
    DosCreateMutexSem(
                        NULL,
                                             /* Local semaphore, no name */
                                             /* Addr of semaphore handle */
                        &lock,
                                             /* No options
                        0.
                                             /* Initially unowned
         Start the threads:
    thread2Tid= _beginthread(
                                             /* Address of function
/* Don't really give stack
                        thread2.
                        NULL,
                                             /* Size of stack needed
                        STACK_SIZE,
                                             /* Message to thread
                        (PVOID) 0);
    thread3Tid= _beginthread(
                        thread3,
                                             /* Address of function
                                             /* Don't really give stack
                                             /* Size of stack needed
                        STACK_SIZE,
                        (PVOID) 0);
                                             /* Message to thread
        Wait for thread 2 to finish
    DosWaitThread(&thread2Tid, DCWW_WAIT);
     * Thread 2's done, kill thread 3
    DosKillThread(thread3Tid);
}
        thread2()
        Thread2 simulates a thread working with a global resource.
    It reads a global structure, then pretends to work with it for
    awhile and reads it again. If the two reads produce different
    results, it complains. Unlike the code in prog7-1, here we
    use a mutual exclusion semaphore to serialize access to the
    global location. This way, we should never get a complaint.
```

```
void thread2(void *foo)
{
   long
                       localValue1:
   long
                       localValue2;
   long
   long
                       tmp;
    * Let the other thread run a bit.
   DosSleep(1000);
    * Now pretend to do useful work
   for (i = 0; i < 100000; i++) {
        * Request the mutex semaphore, locking the resource.
       DosRequestMutexSem(
               lock,
                                       /* Mutex semaphore
               SEM_INDEFINITE_WAIT); /* Wait till we get it
        * Read the global resource
       localValue1= resource.value1;
        * Act busy
       for (tmp= 0; tmp < 10; tmp++) {
        * Read the resource again.
       localValue2= resource.value2;
        * We're done messing with it. Release it so others can use it.
       DosReleaseMutexSem(lock);
        * Now see if the two reads produced the same result
       if (localValue1 != localValue2) {
           printf("The resource was corrupted\n"
                  " localValue1= %d, localValue2= %d\n",
                  localValue1, localValue2);
           return;
           }
       }
   /*
```

Figure 7.2 Continued

```
* A miracle ! We made it !
   printf("Thread ran to completion successfully\n");
        thread3()
       Thread3 increments the values in the global structure
    "resource" until it is killed. Here, we need to get ownership
    of the mutual exclusion semaphore before we can modify the
    structure.
void thread3(void *foo)
    for (;;) {
           Request the mutex semaphore, locking the resource.
       DosRequestMutexSem(
                lock,
                                           Mutex semaphore
                SEM_INDEFINITE_WAIT); /* Wait till we get it
        resource.value1++;
        resource.value2++;
           We're done messing with it. Release it so others can use it.
        DosReleaseMutexSem(lock);
```

Program prog7-3.c, which is presented in Figure 7.3, shows how timeouts can be used to return from a DosRequestMutexSem(). Thread 3 hogs the semaphore, causing thread 2's request to time out. On a timeout return, the calling thread does not own the semaphore.

Figure 7.3 prog7-.3.c

```
prog7-3.c
    This program demonstrates the use of timeouts when waiting
for ownership of a semaphore.
    Thread 2 attempts to gain ownership of a semaphore. It
requests a timeout after two seconds. Thread 3 hogs the
semaphore for 10 seconds. The request times out.
```

```
#include <stdlib.h>
#include <process.h>
#define INCL_DOSERRORS
#define INCL DOSPROCESS
#define INCL_DOSSEMAPHORES
#include <os2.h>
#include "mt.h"
   Function declarations for the threads.
void thread2(void *);
void thread3(void *):
/
* Declare our Mutual Exclusion semaphore
HMTX
                        lock;
/

* Declare the global "resource"
lona
                        resource= 0:
        Main()
void main(int argc, char *argv[])
{
    TID
                         thread2Tid;
   TID
                         thread3Tid;
    /*
* Create the mutual exclusion semaphore
   DosCreateMutexSem(
                        NULL,
                                             /* Local semaphore, no name */
                        &lock,
                                             /* Addr of semaphore handle */
                                            /* No options
                                             /* Initially unowned
                        FALSE);
        Start the threads:
    thread2Tid= _beginthread(
                        thread2,
                                             /* Address of function
                                             /* Don't really give stack
                        NULL,
                        STACK_SIZE,
                                             /* Size of stack needed
                                             /* Message to thread
                        (PVOID) 0);
    thread3Tid= _beginthread(
                        thread3,
                                             /* Address of function
                                            /* Don't really give stack
/* Size of stack needed
                        NULL,
                        STACK SIZE.
                        (PVOID) 0):
                                            /* Message to thread
       Wait for thread 2 to finish
```

#include <stdio.h>

Figure 7.3 Continued

```
DosWaitThread(&thread2Tid, DCWW_WAIT);
    * Thread 2's done, kill thread 3
    DosKillThread(thread3Tid);
}
        thread2()
        Thread2 requests a mutex semaphore timing out after 2
    seconds.
void thread2(void *foo)
    ULONG
                        err;
    while (TRUE) {
        if (err= DosRequestMutexSem(
                    lock,
                                        /* Semaphore handle
                    2000)) {
                                        /* Only wait 2 seconds
            switch (err) {
            case ERROR_TIMEOUT:
                printf("Request for lock timed out."
                       " We do not have resource\n");
                break;
            case ERROR_SEM_OWNER_DIED:
                printf("Semaphore owner died. We do not have resource\n");
            default:
                printf("Error: DosRequestMutexSem(), #%d\n", err);
                break;
                }
            return;
        DosReleaseMutexSem(lock);
        thread3()
        Thread3 holds the semaphore for 10 seconds, causing thread 2
    to time out.
void thread3(void *foo)
    for (;;) {
        DosRequestMutexSem(
```

```
lock, /* Mutex semaphore *,
SEM_INDEFINITE_WAIT); /* Wait till we get it *,
DosSleep(10000);

DosReleaseMutexSem(lock);
}
```

Program prog7-4.c, which is presented in Figure 7.4, shows the error return from a DosRequestMutexSem() when the owner thread terminates. Note that the calling thread does not own the semaphore. All waiters are awakened with this error.

Figure 7.4 prog7-4.c

```
prog7-.c
        This program demonstrates a DosRequestMutexSem() call
    returning an error when the semaphore owner dies.
        Thread 2 attempts to gain ownership of a semaphore. It
    requests a timeout after two seconds. Thread 3 gets ownership,
    then terminates. Thread 2 is woken with a ERROR_SEM_OWNER_DIED
#include <stdio.h>
#include <stdlib.h>
#include <process.h>
#define INCL DOSERRORS
#define INCL_DOSPROCESS
#define INCL_DOSSEMAPHORES
#include <os2.h>
#include "mt.h"
   Function declarations for the threads.
void thread2(void *);
void thread3(void *);
  Declare our Mutual Exclusion semaphore
HMTX
                        lock;
   Delcare the global "resource"
long
                        resource= 0;
       Main()
```

Figure 7.4 Continued

```
void main(int argc, char *argv[])
    TID
                         thread2Tid:
    TID
                         thread3Tid;
        Create the mutual exclusion semaphore
    DosCreateMutexSem(
                         NULL,
                                               /* Local semaphore, no name */
                                               /* Addr of semaphore handle */
                         &lock,
                         0,
                                               /* No options
                         FALSE):
                                              /* Initially unowned
         Start the threads:
    thread2Tid= _beginthread(
                         thread2,
                                              /* Address of function
                                              /* Don't really give stack
/* Size of stack needed
                         STACK_SIZE,
                         (PVOID) 0);
                                               /* Message to thread
    thread3Tid= _beginthread(
                         thread3,
                                               /* Address of function
                                               /* Don't really give stack
                         NULL,
                                              /* Size of stack needed
/* Message to thread
                         STACK_SIZE,
                         (PVOID) 0);
        Wait for thread 2 to finish
    DosWaitThread(&thread2Tid, DCWW_WAIT);
        Thread 2's done, kill thread 3
    DosKillThread(thread3Tid);
}
        thread2()
        Thread2 requests a mutex semaphore timing out after 2
    seconds.
void thread2(void *foo)
    ULONG
                         err;
    while (TRUE) {
        if (err= DosRequestMutexSem(
                     lock,
                                           /* Semaphore handle
                     2000)) {
                                           /* Only wait 2 seconds
             switch (err) {
             case ERROR_TIMEOUT:
```

```
printf("Request for lock timed out."
                       " We do not have resource\n");
                break:
            case ERROR SEM OWNER DIED:
                printf("Semaphore owner died. We do not have resource\n");
                break;
            default:
                printf("Error: DosRequestMutexSem(), #%d\n", err);
                break:
            return;
        DosReleaseMutexSem(lock);
}
        thread3()
        Thread3 holds the semaphore for 10 seconds, causing thread 2
    to time out.
void thread3(void *foo)
{
    for (;;) {
       DosRequestMutexSem(
                                        /* Mutex semaphore
                lock.
                SEM_INDEFINITE_WAIT);
                                        /* Wait till we get it
           We return, which will cause the thread to terminate.
           Because we own the semaphore, any threads waiting for
           ownership will be notified that we died.
        */
        return:
       DosReleaseMutexSem(lock);
}
```

Summary

Mutex semaphores are used to serialize access to resources. A resource can be as simple as an integer variable or as complex as a disk file. For a mutex semaphore to be effective, all threads that access the resource that it protects must request the semaphore before accessing the resource.

DosCloseMutexSem()

The ${\tt DosCloseMutexSem}()$ function ends access to a mutex semaphore for all the threads in a process.

Prototype

Parameter

hmtx Holds the mutex semaphore handle to close.

Return value

0 NO_ERROR

6 ERROR_INVALID_HANDLE

301 ERROR_SEM_BUSY

Description

The DosCloseMutexSem() function releases access to a mutex semaphore. When all access to a semaphore has been released, the semaphore itself is deleted.

DosCreateMutexSem()

The DosCreateMutexSem() function creates a mutual exclusion semaphore and opens it for use by the current process.

Prototype

Parameters

Pointer to a null terminated string with the name of the semaphore to create, or NULL to create an unnamed semaphore. A semaphore name must begin with the prefix \SEM32\.
 Pointer to the location where the handle of the new semaphore will be returned.
 flattr Mutex semaphore creation attributes. DC_SEM_SHARED makes an un-

fstate Initial state of the mutex semaphore. If it is FALSE, the initial state is "not owned." If it is TRUE, the initial state is "owned by the creating thread."

Return value

0 NO_ERROR

8 ERROR_NOT_ENOUGH_MEMORY

named semaphore shared.

87 ERROR_INVALID_PARAMETER

123 ERROR INVALID NAME

285 ERROR_DUPLICATE_NAME

290 ERROR_TOO_MANY_HANDLES

Description

The DosCreateMutexSem() function creates a mutual exclusion (mutex) semaphore and opens it for use by the current process. The semaphore can be either named or unnamed depending on whether <code>pszName</code> specifies a string or is NULL. A semaphore can be made accessible by the creator or other processes via a call to <code>DosOpenMutexSem()</code>, although it is not necessary for the creator to explicitly open the semaphore. An unnamed semaphore can be created as shared by setting the <code>DC_SEM_SHARED</code> bit in the flAttr parameter. Shared semaphores can be opened by

86 OS/2 Multithreading Facilities

processes other than the creator. Note that, because a shared, unnamed semaphore has no name, its handle must, in some way, be made available by the creator to processes that need to use it. All named semaphores are shared, so it is not necessary to explicitly set the DC_SEM_SHARED bit when creating a named semaphore.

DosOpenMutexSem()

The DosOpenMutexSem() function opens a mutex semaphore.

Prototype

APIRET DosOpenMutexSem(

PSZ pszName, PHMTX phmtx);

Parameters

pszName Pointer to the null terminated string holding the name of the semaphore

to open or NULL, if the semaphore handle is provided.

phmtx Pointer to the location holding semaphore handle. If pszName is non-

NULL, phmtx must be initialized to zero and the handle is returned. If pszName is NULL, phmtx must be initialized to the handle of semaphore

to open.

Return value

0 NO_ERROR

6 ERROR_INVALID_HANDLE

87 ERROR INVALID PARAMETER

105 ERROR_SEM_OWNER_DIED

123 ERROR_INVALID_NAME

187 ERROR_SEM_NOT_FOUND

291 ERROR TOO MANY OPENS

Description

The DosOpenMutexSem() function opens a mutex semaphore for use by a process. Named semaphores can be opened by name or handle, unnamed semaphores must be opened by handle.

DosQueryMutexSem()

Function DosQueryMutexSem() retrieves information about the current state of a mutex semaphore.

Prototype

Parameters

hmtx The handle of the semaphore to query.

ppid A pointer to the location in which the process ID of the owner will be

returned.

ptid A pointer to the location in which the thread ID of the owner will be

returned

pulCount A pointer to the location where the request nesting count will be

returned.

Return value

0 NO_ERROR

6 ERROR INVALID HANDLE

87 ERROR_INVALID_PARAMETER

105 ERROR_SEM_OWNER_DIED

Description

Function DosQueryMutexSem() retrieves information about the current state of a mutex semaphore. The owner (if any) of the semaphore is returned as a PID/TID pair indicating the process ID of the owning thread and thread ID of the owning thread. In addition, the request count is returned. The request count is incremented when a DosRequestMutexSem() is issued by the owning thread and decremented when a DosReleaseMutexSem() is issued by the owning thread. Thus, a count of zero indicates that the semaphore is unowned, a nonzero value indicates that the semaphore is owned. Request count gives the number of DosReleaseMutexSem() calls that must be issued by the owner of the semaphore before it actually is released.

DosReleaseMutexSem()

Function DosReleaseMutexSem() releases ownership of a mutex semaphore that was requested by the DosRequestMutexSem() function.

Prototype

Parameter

hmtx The handle of the mutex semaphore to release.

Return value

```
0 NO_ERROR
6 ERROR_INVALID_HANDLE
288 ERROR_NOT_OWNER
```

Description

Function DosReleaseMutexSem() releases ownership of a mutex semaphore that was requested by the DosRequestMutexSem() function. If multiple calls have been made to DosRequestMutexSem() by the owning thread, then the same number of calls are required to DosReleaseMutexSem() to fully release the semaphore. Note that this call can be issued only by the thread that owns the mutex semaphore.

DosRequestMutexSem()

The DosRequestMutexSem() function requests ownership of a mutual exclusion semaphore.

Prototype

#define INCL_DOSSEMAPHORES

#include <os2.h>

APIRET DosRequestMutexSem(

ULONG

hmtx,
ulTimeout);

Parameters

hmtx The handle of the mutex semaphore being requested.

ulTimeout Sets the time out in milliseconds or 0 for SEM_IMMEDIATE_RETURN or

-1 for SEM_INDEFINITE_WAIT.

Return value

0 NO_ERROR

6 ERROR_INVALID_HANDLE

95 ERROR INTERRUPT

103 ERROR_TOO_MANY_SEM_REQUESTS

105 ERROR_SEM_OWNER_DIED

640 ERROR_TIMEOUT

Description

This function is called to request ownership of a mutual exclusion semaphore. Ownership is relinquished via a call to DosReleaseMutexSem(). Calls to Dos RequestMutexSem() can be nested: a count is kept of the number of times that Dos

RequestMutexSem() has been called by the owning thread, and that number of calls to DosReleaseMutexSem() are required before another thread will be granted

ownership. Unless SEM_IMMEDIATE_RETURN is specified, this call will block until ownership can be passed to the current thread. If a timeout is specified, the call will return with an error if the timeout elapses before ownership can be obtained.

Chapter

8

Event Semaphores

Where mutual exclusion semaphores are used for mediating access to shared resources, event semaphores are used for virtually every other type of thread synchronization. It is even possible to implement mutual exclusion semaphores using event semaphores.

The best way to think of event semaphores is as traffic lights. An event semaphores has two basic states: posted, which corresponds to a green light, and reset, which corresponds to a red light. When a thread calls <code>DosWaitEventSem()</code>, if the semaphore is in the posted state, the call returns and, if the semaphore is the reset state, the call blocks until the semaphore is posted. When the semaphore is posted, everyone who has been waiting is unblocked. When you come to a traffic light, if it is green you keep going; however, if it is red, you wait until it turns green. It's as simple as that. Well, maybe not.

The difference here is that in addition to stopping and going, you also have to turn the light red at the right time and turn it green at the right time. Turning the light green corresponds to posting to the semaphore using <code>DosPostEventSem()</code>. Turning the light red corresponds to resetting the semaphore using <code>DosResetEventSem()</code>. If a thread is waiting on an event semaphore, and the semaphore is posted and reset before that thread has an opportunity to run, it still will be ready-to-run. Thus, it is possible to have the semaphore reset, or red, immediately upon return from the <code>DosWaitEventSem()</code>.

A semaphore keeps a count of the number of times that it was posted to since the last time it was reset. This value is returned by the DosResetEventSem() call. This value can be useful when the semaphore is being used to communicate discrete events.

Shared semaphores can be accessed by processes other than the creator of the semaphore. For other processes to access a semaphore, they must use DosOpen EventSem() supplying either the name of the semaphore (for a named semaphore) or the semaphore handle (for an unnamed shared semaphore).

Program prog8-1.c, which is presented in Figure 8.1, shows very clearly how event semaphores resemble traffic lights. There is one event semaphore, eventSem. Four threads are created that loop waiting on this semaphore and printing out a message. The main thread starts and stops the other four threads by posting and resetting the event semaphore. Note that even after it issues the DosResetEventSem(), the main thread cannot be certain that the other threads will not run. It is possible that one or more of them have just passed the DosWaitEventSem() and will execute one more iteration before begin stopped by the next wait. In chapter 12, we will show a program that does similar stop-and-go synchronization but such that the controlling thread can be sure when the other threads are stopped.

Figure 8.1 prog8-1.c

```
prog8-1.c
        This program demonstrates the use of event semaphores to
    start and stop other tasks.
        Threads 2 through 5 cycle, waiting for eventSem and printing
    a message. Thread 1 posts to the semaphore allowing them to
    run. Because they do not reset the semaphore, they run until
    thread 1 resets the semaphore.
#include <stdio.h>
#include <stdlib.h>
#include cess.h>
#define INCL_DOSPROCESS
#define INCL DOSSEMAPHORES
#include <os2.h>
#include "mt.h"
    Function declarations for the threads.
void threadN(void *);
   Declare our Event semaphore
HEV
                        eventSem;
    Declare a global that will indicate the stage we're at
long
                        stage;
        Main()
void main(int argc, char *argv[])
    TID
                        thread[4];
    ULONG
                        i;
    ULONG
                        eventCount;
```

```
* Create the event semaphore
    DosCreateEventSem(
                                                 /* Local semaphore, no name */
                          NULL.
                          &eventSem,
                                                 /* Addr of semaphore handle */
                                                 /* No options
                                                 /* Initially not posted
                          FALSE);
        Create the threads
    for (i = 0; i < 4; i++) {
         thread[i]= _beginthread(
                                               /* Address of function
/* Don't really give stack
/* Size of stack needed
/* Message to thread
                          threadN,
                          NULL.
                          STACK SIZE,
                          (PVOID) (i+2));
    /*
* Wait a little while
    DosSleep(1000);
    for (stage= 1; stage <= 5; stage++) {</pre>
        ^{\prime\ast} Post to the semaphore allowing the threads to run. ^{\ast\prime}
        DosPostEventSem(eventSem);
         * Let them run for a little bit
*/
        DosSleep(400);
        /*

* And stop any movement
         DosResetEventSem(eventSem, &eventCount);
          * Wait awhile
        DosSleep(2000);
        Print final message
    printf("We're done.\n");
         threadN()
         ThreadN repeatedly prints a message as long as eventSem is
    is a posted state. When thread 1 resets it, this routine will
    become blocked when it gets to the DosWaitEventSem() call.
void threadN(void *threadNum)
    long
                          me;
```

Figure 8.1 Continued

In program prog8-2.c, which is presented in Figure 8.2, we present a more typical use of event semaphores to signal discrete events. Where in the previous program the semaphore was posted and reset by one thread and waited on by another, in this program, the semaphore is posted by one thread and waited on and reset by the other. This flexibility in when and where a semaphore is reset is what gives OS/2's event semaphores their power. In this program, the events that are being signalled are keystrokes. Note that exactly one DosPostEventSem() is issued for each keystroke. When the semaphore is reset, the post count is used to determine the number of keystrokes that have occurred since the last reset. Notice that the only shared database is the ring buffer into which the characters are placed. Each thread keeps its own pointer into the ring buffer. Thread 1, which reads from the keyboard and adds characters to the ring buffer, uses the semaphore to signal to thread 2 how many characters are available. Note that no attempt has been made to deal with ring buffer overflow. If more than RINGBUFF SIZE keys are hit before thread 2 can process them, some keys will be overwritten. This problem could be solved with a second semaphore, which causes thread 1 to block until there is space in the ring buffer, but this would complicate the program.

Figure 8.2 prog8-2.c

prog8-2.c This program demonstrates using event semaphores to signal events. In this program, keystrokes are passed from thread 1 to thread 2 in a ring buffer. In order to synchronize the threads an event semaphore, keystrokeSem, is used. When a key is put into the ring buffer the semaphore is posted. They echo task waits on the semaphore. When it wakes up, it resets the semaphore, getting the count of the number of times the semaphore was posted as well as putting it in a nonposted state so that a wait will not return until another post occurs. The number of times posted corresponds to the number of keystrokes put into the buffer. Note that there is no protection against buffer overflow. If more than RINGBUG_SIZE keystrokes occur before thread 2 wakes up, late keystrokes will overwrite early keystrokes. This could be corrected with another semaphore, but we want to keep the example as simple as possible.

```
#include <stdio.h>
#include <stdlib.h>
#include <process.h>
#define INCL_KBD
#define INCL_DOSPROCESS
#define INCL_DOSSEMAPHORES
#include <os2.h>
#include "mt.h"
/*

* Define the size of the ring buffer
#define RINGBUFF_SIZE
/*

* Function declarations for the threads.
void thread2(void *);
* Declare our Event semaphore
HEV
                        keystrokeSem;
* Declare a global ring buffer
char
                        ringBuff[RINGBUFF_SIZE];
       Main()
void main(int argc, char *argv[])
{
    TID
                        thread2Tid;
    ULONG
                       ringBuffOffset;
   HKBD
                        keyboard;
    KBDKEYINFO
                        keyInfo;
    char
      Create the event semaphore
    DosCreateEventSem(
                        NULL,
                                            /* Local semaphore, no name */
                                            /* Addr of semaphore handle */
                        &keystrokeSem,
                                            /* No options
                        FALSE):
                                            /* Initially not posted
       Start the second thread
    thread2Tid= _beginthread(
                        thread2,
                                            /* Address of function
                                           /* Don't really give stack
/* Size of stack needed
                        NULL,
                        STACK SIZE,
                                           /* Message to thread
                        (PVOID) 0);
       Get access to the keyboard
```

Figure 8.2 Continued

```
KbdOpen(&keyboard);
   KbdGetFocus(IO_WAIT, keyboard);
       Get characters and put them into the ring buffer
   c=0;
    ringBuffOffset= 0;
   while (c != "z") {
       KbdCharIn(&keyInfo, IO_WAIT, keyboard);
       c= keyInfo.chChar;
           We've got the character. Put it in the ring buffer.
       ringBuff[ringBuffOffset]= c;
        ringBuffOffset++;
        if (ringBuffOffset >= RINGBUFF_SIZE) {
            ringBuffOffset= 0;
           Post the semaphore to indicate we got another keystroke.
       DosPostEventSem(keystrokeSem);
       Print final message
    printf("We're done.\n");
        thread2()
        Thread2 waits on keystrokeSem. When it wakes up, it resets
    the semaphore, getting the post count. The post count is used
    to determine the number of characters to remove from the ring
    buffer.
void thread2(void *foo)
    ULONG
                        ringBuffOffset;
    ULONG
                        numKeystrokes;
    ringBuffOffset= 0;
    for (;;) {
            Wait for the semaphore to be posted
        DosWaitEventSem(
                                                 /* Semaphore handle
                        keystrokeSem,
                                                /* Don't time out
                        SEM INDEFINITE WAIT);
```

```
Now reset the semaphore and get the number of times
           it was posted.
        */
        DosResetEventSem(
                                                                        */
                        kevstrokeSem.
                                            /* Semaphore handle
                        &numKeystrokes); /* Number of posts
           Print out what we got
        printf("%2d keystrokes received: \"", numKeystrokes);
       while (numKeystrokes) {
           putchar(ringBuff[ringBuffOffset]);
            ringBuffOffset++;
            if (ringBuffOffset >= RINGBUFF SIZE) {
               ringBuffOffset= 0;
            numKeystrokes";
       printf("\"\n");
        * Pause a while to allow for some typeahead
       DosSleep(1000);
}
```

Summary

Event semaphores are used for one thread to signal another thread that a particular event has occurred. They are the workhorse of thread synchronization. Event semaphores work like traffic lights. They have two states: posted and not posted. When in the posted state, waits on the semaphore will not block. When in the reset state, waits on the semaphore will block until the semaphore is posted.

DosCloseEventSem()

The DosCloseEventSem() function ends access to an event semaphore for all the threads in a process.

Prototype

Parameter

hev The handle of the event semaphore to close.

Return value

- 0 NO_ERROR
- 6 ERROR_INVALID_HANDLE
- 301 ERROR_SEM_BUSY

Description

The DosCloseEventSem() function releases access to an event semaphore. When all access to a semaphore has been released, the semaphore itself is deleted.

DosCreateEventSem()

The DosCreateEventSem() function creates an event semaphore and opens it for use by all of the threads in the current process.

Prototype

Parameters

pszName A pointer to a null terminated string with the name of the semaphore to create, or NULL to create an unnamed semaphore. A semaphore name must begin with the prefix \SEM32\.

phev A pointer to the location where the handle of the new semaphore will

be returned.

flattr The event semaphore creation attributes. DC_SEM_SHARED makes un-

named semaphore shared.

fState The initial state of the event semaphore. If it is FALSE, the initial state

is "not posted." If it is TRUE, initial state is "posted."

Return value

0 NO ERROR

8 ERROR_NOT_ENOUGH_MEMORY

87 ERROR_INVALID_PARAMETER

123 ERROR INVALID NAME

285 ERROR_DUPLICATE_NAME

290 ERROR TOO MANY HANDLES

Description

The DosCreateEventSem() function creates an event semaphore and opens it for use by the current process. A semaphore can be made accessible by the creator or other processes via a call to DosOpenEventSem(), although it is not necessary for the creator to explicitly open the semaphore. An unnamed semaphore can be created as shared by setting the DC_SEM_SHARED bit in the flattr parameter. Shared semaphores can be opened by processes other than the creator. Note that, because a shared, unnamed semaphore has no name, its handle must, in some way, be made available by the creator to processes that need to use it. All named semaphores are shared, so it is not necessary to explicitly set the DC_SEM_SHARED bit when creating a named semaphore.

DosOpenEventSem()

The ${\tt DosOpenEventSem}$ () function opens an event semaphore and returns a handle to it.

Prototype

Parameters

pszName A pointer to the null terminated string holding the name of the semaphore to open or NULL, if the semaphore handle is provided.

phev A pointer to the location holding semaphore handle. If pszName is non-NULL, phev must be initialized to zero and the handle is returned. If pszName is NULL, phev must be initialized to handle of semaphore to open.

Return value

- 0 NO ERROR
- 6 ERROR_INVALID_HANDLE
- 8 ERROR SEM BUSY
- 87 ERROR INVALID PARAMETER
- 123 ERROR_INVALID_NAME
- 187 ERROR_SEM_NOT_FOUND
- 291 ERROR TOO MANY OPENS

Description

The DosOpenEventSem() function opens an event semaphore for use by a process. Named semaphores can be opened by name or handle, unnamed semaphores must be opened by handle.

DosPostEventSem()

Function DosPostEventSem() posts an event semaphore. This allows threads waiting on the semaphore to run.

Prototype

Parameter

hev Holds the event semaphore handle.

Return value

- 0 NO_ERROR
- 6 ERROR INVALID HANDLE
- 298 ERROR TOO MANY POSTS
- 299 ERROR_ALREADY_POSTED

Description

The DosPostEventSem() function posts an event semaphore. If the semaphore was reset, it becomes posted with a post count of 1. If it already was posted, the post count is incremented. If any threads were waiting on the semaphore using a DosWaitEventSem() call, they are unblocked. Threads that are unblocked in this manner will not be reblocked if the semaphore is reset before they have been able to run. The semaphore will remain in a posted state, allowing threads to pass through DosWaitEventSem() calls without blocking until it is reset.

DosQueryEventSem()

Function ${\tt DosQueryEventSem}()$ retrieves the information about the current event semaphore owner.

Prototype

Parameters

hev The handle of the event semaphore.

pulPostCt A pointer to a location that will receive the number of times that

DosPostEventSem() has been called since the last time the sema-

phore was reset.

Return value

- 0 NO ERROR
- 6 ERROR INVALID HANDLE
- 87 ERROR_INVALID_PARAMETER

Description

Function DosQueryEventSem() retrieves the post count of an event semaphore. The post count is the number of times the semaphore has been posted since the last reset.

DosResetEventSem()

Function DosResetEventSem() puts a semaphore into the reset state, causing all threads that subsequently call DosWaitEventSem() to block.

Prototype

```
#define INCL_DOSSEMAPHORES
#include <os2.h>

APIRET DosResetEventSem(
    HEV hev,
    PULONG pulPostCt);
```

Parameters

hev The event semaphore handle.

pulPostCt The address of a location that receives the number of times that

DosPostEventSem() has been called since the last time the sema-

phore was in the reset state.

Return value

- 0 NO ERROR
- 6 ERROR_INVALID_HANDLE
- 87 ERROR_ALREADY_RESET

Description

Function DosResetEventSem() puts a semaphore into a reset state and returns its post count. While it is in the reset state, calls to DosWaitEventSem() will block until the semaphore is posted.

DosWaitEventSem()

The DosWaitEventSem() function waits for an event semaphore to be posted.

Prototype

```
#define INCL_DOSSEMAPHORES
#include
          <os2.h>
APIRET DosWaitEventSem(
     HEV
            hev.
      ULONG ulTimeout):
```

Parameters

hev

The event semaphore handle.

ulTimeout Sets time out in milliseconds or 0 for SEM IMMEDIATE RETURN or -1 for SEM INDEFINITE WAIT.

Return value

- 0 NO ERROR
- 6 ERROR INVALID HANDLE
- 8 ERROR NOT ENOUGH MEMORY
- 95 ERROR INTERRUPT
- 640 ERROR_TIMEOUT

Description

The DosWaitEventSem() function blocks the caller if the specified semaphore is in the reset state. The thread will become unblocked when the semaphore is posted. If the semaphore is in a posted state when the DosWaitEventSem() call is issued, the call will return immediately. Note that, if the thread blocks on a reset semaphore and the semaphore is posted and reset before the thread has run, it still is in a ready-torun state and will return from the DosWaitEventSem() call.

Unless SEM IMMEDIATE RETURN is specified this call will block until ownership can be passed to the current thread. If a timeout is specified the call will return with an error if the timeout elapses before the semaphore has been posted.

Chapter

9

Multiple Wait Semaphores

Multiple wait semaphores, or muxwait semaphores, allow a thread to wait for several events in one call. A *muxwait semaphore* is a conglomeration of either event or mutex semaphores. There are essentially four types of muxwait semaphores:

- Any event
- All event
- Any mutex
- All mutex

A muxwait semaphore is created with a list of event of semaphores. The list must consist of all event semaphores or all mutex semaphores; event and mutex semaphores cannot be mixed.

Waiting on a muxwait semaphore that has been created as "any event" will cause the thread to block until one of the events in the semaphore list associated with the muxwait semaphore is posted.

A muxwait semaphore that has been created as "all event" will wait for all of the events in the semaphore list to become posted. Note that they must all be in a posted state at the same time. Suppose that muxwait semaphore A consists of two event semaphores 1 and 2 and that thread X is waiting on it. If event 1 is posted and reset and event 2 is posted and reset, thread X will not be unblocked. It will remain blocked until both semaphores 1 and 2 are in a posted state at the same time.

Waiting on a muxwait semaphore that has been created as "any mutex" will cause the thread to block until one of the mutex semaphores in the list associated with the muxwait semaphore to be released.

If a muxwait semaphore is created as "all mutex" waiting threads do not unblock until all of the mutex semaphores in the list become free at the same time. As in the case of "all event" muxwait semaphores, the "all mutex" semaphore will not unblock if one or more of the semaphores in its list are owned at any given time. Thus, a thread trying to gain ownership of a set of resources can be "starved" if other threads are always holding some of the desired resources.

Shared semaphores can be accessed by processes other than the creator of the semaphore. For other processes to access a semaphore, they must use DosOpenMux WaitSem() supplying either the name of the semaphore (for a named semaphore) or the semaphore handle (for an unnamed shared semaphore).

Program prog9-1.c, which is presented in Figure 9.1, demonstrates the operation of an "all mutex" muxwait semaphore. Three mutex semaphores are created, then the muxwait semaphore is created. Three threads are created. Each of the threads obtains and releases one of the three mutex semaphores. The main thread waits on the muxwait semaphore. When all three mutex semaphores become available, it unblocks and gets ownership of the three semaphores. We can see the starvation problem if we remove the <code>DosSleep()</code> at the end of the loop in <code>threadN()</code>. This will cause the threads to almost always own at least one of the respective semaphores, making it extremely unlikely that all three semaphores will become available at the same time for the main thread.

Figure 9.1 prog9-1.c

```
prog9-1.c
        This program demonstrates the use of Multiplexed semaphores.
        Three threads are created, each of which holds and releases
   a mutex semaphore. A mux semaphore is made that requires all
    three mutex semaphores. Thread 1 waits on the mux semaphore.
#include <stdio.h>
#include <stdlib.h>
#include <process.h>
#define INCL_DOSPROCESS
#define INCL_DOSSEMAPHORES
#include <os2.h>
#include "mt.h"
   Function declarations for the threads.
void threadN(void *);
   Declare our Event semaphore
*/
HMTX
                        resource[3];
HMUX
                        allResources;
        Main()
```

```
void main(int argc, char *argv[])
    ULONG
   ULONG
                       lastResource:
    TID
                        thread[3];
    SEMRECORD
                       resourceList[3];
       Create three mutex semaphores
   DosCreateMutexSem(
               NULL.
                                            /* Local semaphore, no name */
               &resource[0],
                                            /* Addr of semaphore handle */
                                            /* No options
               FALSE);
                                            /* Initially not posted
   DosCreateMutexSem(
               NULL.
                                            /* Local semaphore, no name */
                                            /* Addr of semaphore handle */
               &resource[1],
                                            /* No options
               FALSE):
                                           /* Initially not posted
   DosCreateMutexSem(
               NULL.
                                            /* Local semaphore, no name */
               &resource[2],
                                            /* Addr of semaphore handle */
                                           /* No options
                                            /* Initially not posted
               FALSE);
       Now set up a semaphore list for the DosCreateMuxSem()
    resourceList[0].hsemCur= (HSEM) resource[0];
    resourceList[1].hsemCur= (HSEM) resource[1];
    resourceList[2].hsemCur= (HSEM) resource[2];
       Create the Mux semaphore.
   DosCreateMuxWaitSem(
                                           /* Local semaphore, no name */
                                            /* Addr of semaphore handle */
                       &allResources.
                                           /* Three mutex semaphores */
                                           /* Array of semaphores
                       resourceList.
                       DCMW_WAIT_ALL);
                                           /* Wait till all are free
       Start the threads
    for (i = 0; i < 3; i++) {
        thread[i] = _beginthread(
                                            /* Address of function
                       threadN,
                                           /* Don't really give stack
                       NULL,
                                           /* Size of stack needed
                       STACK SIZE.
                        (PVOID) i);
                                           /* Message to thread
       }
    for (i = 0; i < 4; i++) {
        * Wait for the Mux semaphore. We will not waken till we have
          all of the resources
       DosWaitMuxWaitSem(allResources, SEM_INDEFINITE_WAIT, &lastResource);
        * We've got all three semaphores
```

{

Figure 9.1 Continued

```
printf(ANSI_RED " R0 R1 R2\n" ANSI_RESET);
        /*
* Hold then awhile
       DosSleep(1000);
        * Now release them.
        DosReleaseMutexSem(resource[0]);
        DosReleaseMutexSem(resource[1]);
       DosReleaseMutexSem(resource[2]);
           And display that fact.
        printf(ANSI_GREEN "
                              RO R1 R2\n" ANSI_RESET);
}
        threadN()
        ThreadN gets resource n and holds it for a little while,
    then releases it.
void threadN(void *meN)
{
    ULONG
                       me;
    long
                       i;
                       indent[33];
    char
    me= (ULONG) meN;
    i = (me + 1) * 8;
    indent[i]= 0;
    for (i"; i >= 0; i") {
        indent[i]= " ";
    for (;;) {
       /*
* Get our semaphore
        DosRequestMutexSem(
                        resource[me], /* Semaphore handle
SEM_INDEFINITE_WAIT); /* Don't time out
        /*
* Print message that we got semaphore
        printf(ANSI_RED "%sR%d\n" ANSI_RESET, indent, me);
           Wait a little bit
```

```
*/
DosSleep(200);

/*
    * Now release the semaphore
    */
DosReleaseMutexSem(resource[me]);

/*
    * Print message that it's released.
    */
printf(ANSI_GREEN "%sR%d\n" ANSI_RESET, indent, me);

/*
    * Wait for awhile before starting again.
    * If we comment out this delay, we will see that a muxwait
    * semaphore can be starved: as long as one of the semaphores
    * is owned at any time, the muxwait will never return.
    * This pause ensures that each semaphore generally is free
    * making it highly likely that the muxwait will be able to get
    * all three.
    */
DosSleep(2000);
}
```

Program prog9-2.c, which is presented in Figure 9.2, demonstrates the operation of an "all event" semaphore. Two event semaphores are created, and a muxwait semaphore is created based on these semaphores. Two threads are created. One thread waits for keystrokes and posts one of the event semaphores when a keystroke occurs. The other thread waits for mouse events and posts to the other event semaphore when a button 1 click occurs. The main thread waits on the muxwait semaphore. It unblocks after both a button click and a keystroke have occurred. It then resets the semaphores and waits again.

Figure 9.2 prog9-2.c

Figure 9.2 Continued

```
#define INCL_MOU
#define INCL_DOSPROCESS
#define INCL_DOSSEMAPHORES
#include <os2.h>
#include "mt.h"
    Function declarations for the threads.
void mouseThread(void *);
void keyboardThread(void *);
    Declare our Event semaphore
HEV
                         mouseClickSem;
HEV
                         keystrokeSem;
HMUX
                         bothEvents:
        Main()
void main(int argc, char *argv[])
    ULONG
                         lastEvent:
    ULONG
                         mouseCount;
    ULONG
                         keystrokeCount;
    TID
                         mouseThreadTid;
    TID
                         keyboardThreadTid;
    SEMRECORD
                         semaphoreList[2];
     * Create the two semaphores: one for the keyboard and one
     ^{\star} for the mouse.
    DosCreateEventSem(
                         NULL,
                                        /* Local semaphore, no name */
                         &mouseClickSem, /* Addr of semaphore handle */ 0, /* No options */
                                         /* Initially not posted
                         FALSE);
    DosCreateEventSem(
                                         /* Local semaphore, no name */
                                        /* Addr of semaphore handle */
                         &keystrokeSem,
                                         /* No options
                         0.
                                         /* Initially not posted
                         FALSE):
        Create the semaphore list for the MuxWait
    semaphoreList[0].hsemCur= (HSEM) mouseClickSem;
    semaphoreList[1].hsemCur= (HSEM) keystrokeSem;
        Create the MuxWait semaphore
    DosCreateMuxWaitSem(
```

```
/* Local semaphore, no name */
                                       /* Addr of semaphore handle */
                        &bothEvents,
                        2, /* Three mutex semaphores */
semaphoreList, /* Array of semaphores */
                        DCMW_WAIT_ALL); /* Wait till all are free
       Start the two threads
    mouseThreadTid= _beginthread(
                        mouseThread,
                                            /* Address of function
                                            /* Don't really give stack
                        NULL,
                                            /* Size of stack needed
                        STACK_SIZE,
                                            /* Message to thread
                        (PVOID) 0);
    keyboardThreadTid= _beginthread(
                                            /* Address of function
                        keyboardThread,
                                            /* Don't really give stack
                        NULL.
                        STACK SIZE,
                                            /* Size of stack needed
                                            /* Message to thread
                        (PVOID) 0):
    for (;;) {
            Wait until both a keystroke and a mouse click have occured
       DosWaitMuxWaitSem(
                                           /* Semaphore handle
                        bothEvents,
                        SEM_INDEFINITE_WAIT,/* Wait forever
                                        /* Last sem ID (unused here)*/
                        &lastEvent);
            Reset the semaphores and get the counts
        DosResetEventSem(mouseClickSem, &mouseCount);
        DosResetEventSem(keystrokeSem, &keystrokeCount);
           Print a message
        printf("Got %d mouse clicks, %d keystrokes\n",
                   mouseCount,
                    keystrokeCount);
        }
}
       mouseThread()
       MouseThread waits for a mouse button 1 click and posts to
   mouseClickSem when it gets one.
void mouseThread(void *foo)
   HMOU
                        mouHand;
   MOUEVENTINFO
                        mouEvent;
   USHORT
                        fWait:
   USHORT
                        eventMask;
```

Figure 9.2 Continued

```
Make the mouse available
   MouOpen(NULL, &mouHand);
   MouDrawPtr(mouHand);
   eventMask= 0x7E;
   MouSetEventMask(&eventMask, mouHand);
   while (TRUE) {
       /*
* Wait for a mouse event
       fWait= 1;
       MouReadEventQue(&mouEvent, &fWait, mouHand);
       if (mouEvent.time) {
            if (mouEvent.fs & MOUSE_BN1_DOWN) {
               printf("Got mouse click\n");
                    Post to the mouse semaphore
               DosPostEventSem(mouseClickSem);
            }
}
        keyboardThread()
        KeyboardThread waits for a keystroke. When it gets one it
   posts to keyboardSem.
void keyboardThread(void *foo)
{
   HKBD
                        keyboard;
    KBDKEYINFO
                        keyInfo;
       Get access to the keyboard
    KbdOpen(&keyboard);
    KbdGetFocus(IO_WAIT, keyboard);
    for (;;) {
            Wait for a keystroke
        KbdCharIn(&keyInfo, IO_WAIT, keyboard);
        printf("Got keystroke\n");
```

```
* Post to the semaphore
    */
    DosPostEventSem(keystrokeSem);
}
```

Program prog9-3.c, which is presented in Figure 9.3, is much like prog9-2 except that it uses an "any event" semaphore. Here, we assign user ID's to the two semaphores when we create the muxwait semaphore. These ID's tell us which event occurred when we return from the DosWaitMuxWaitSem(). Thus, the main thread unblocks when either a keystroke or a mouse click occur. It knows which has occurred because the user id is returned.

Figure 9.3 prog9-3.c

```
prog9-3.c
        This program is similar to prog9-2, except that rather than
    waiting for both events that make up the MuxWait semaphore, it
    waits for any event. It shows how the user ID is used in the
    semaphore list.
#include <stdio.h>
#include <stdlib.h>
#include <process.h>
#define INCL KBD
#define INCL MOU
#define INCL_DOSPROCESS
#define INCL DOSSEMAPHORES
#include <os2.h>
#include "mt.h"
    Function declarations for the threads.
void mouseThread(void *);
void keyboardThread(void *);
    Declare our Event semaphore
 */
HEV
                        mouseClickSem;
HEV
                        keystrokeSem;
HMUX
                        bothEvents:
 * Declare the event types:
typedef enum {
    EVENT_MOUSE_CLICK= 0x1010,
    EVENT_KEYSTROKE= 0x1011
```

Figure 9.3 Continued

```
eventType;
       Main()
void main(int argc, char *argv[])
   ULONG
                        lastEvent:
   ULONG
                        mouseCount;
   ULONG
                        keystrokeCount;
    char
                       *eventString;
   TID
                        mouseThreadTid;
                        keyboardThreadTid;
    TID
    SEMRECORD
                        semaphoreList[2];
       Create the event semaphores
    DosCreateEventSem(
                                        /* Local semaphore, no name */
                        NULL,
                        &mouseClickSem, /* Addr of semaphore handle */
                                       /* No options
                        FALSE):
                                        /* Initially not posted
    DosCreateEventSem(
                                        /* Local semaphore, no name */
                                       /* Addr of semaphore handle */
                        &keystrokeSem,
                                        /* No options
                                        /* Initially not posted
                        FALSE);
       Initialize the semaphore list for the DosCreateMuxWait(). This
       time, we use the ulUser fields to give IDs to the semaphores.
       These IDs are returned when the MuxWait returns.
     */
    semaphoreList[0].hsemCur= (HSEM) mouseClickSem;
    semaphoreList[0].ulUser= EVENT_MOUSE_CLICK;
    semaphoreList[1].hsemCur= (HSEM) keystrokeSem;
    semaphoreList[1].ulUser= EVENT_KEYSTROKE;
    DosCreateMuxWaitSem(
                                        /* Local semaphore, no name */
                                        /* Addr of semaphore handle */
                        &bothEvents,
                                        /* Three mutex semaphores */
                        semaphoreList, /* Array of semaphores
                        DCMW WAIT ANY): /* Wait till ANY are free
       Start the two threads
    mouseThreadTid= _beginthread(
                                             /* Address of function
                        mouseThread.
                                             /* Don't really give stack
                                             /* Size of stack needed
                        STACK_SIZE,
                        (PVOID) 0);
                                             /* Message to thread
    keyboardThreadTid= _beginthread(
                        keyboardThread,
                                             /* Address of function
                        NULL.
                                             /* Don't really give stack
                        STACK_SIZE,
                                            /* Size of stack needed
                                             /* Message to thread
                        (PVOID) 0);
```

```
for (;;) {
        /* Wait until both a keystroke and a mouse click have occured
        DosWaitMuxWaitSem(
                                            /* Semaphore handle
                        bothEvents,
                        SEM_INDEFINITE_WAIT,/* Wait forever
                                          /* Last semaphore ID
                        &lastEvent);
            Last event is the ID that we assigned to the semaphore that
            was signalled. We use it to determine which semaphore
         * it was.
        switch (lastEvent) {
        case EVENT_MOUSE_CLICK:
            DosResetEventSem(mouseClickSem, &mouseCount);
            keystrokeCount= 0;
            eventString= "Mouse Click";
            break;
        case EVENT_KEYSTROKE:
            DosResetEventSem(keystrokeSem, &keystrokeCount);
            mouseCount= 0:
            eventString= "Keystroke";
            break;
        default:
            eventString= "?????";
            break;
        printf("Event was %s. Got %d mouse clicks, %d keystrokes\n",
                    eventString,
                    mouseCount,
                    keystrokeCount);
        }
}
        mouseThread()
       MouseThread waits for a mouse button 1 click and posts to
    mouseClickSem when it gets one.
void mouseThread(void *foo)
{
   HMOU
                        mouHand;
    MOUEVENTINFO
                        mouEvent;
    USHORT
                        fWait:
    USHORT
                        eventMask:
     * Make the mouse available
    MouOpen(NULL, &mouHand);
   MouDrawPtr(mouHand);
    eventMask= 0x7E;
```

Figure 9.3 Continued

```
MouSetEventMask(&eventMask, mouHand);
    while (TRUE) {
           Wait for a mouse event
        fWait= 1;
       MouReadEventQue(&mouEvent, &fWait, mouHand);
        if (mouEvent.time) {
            if (mouEvent.fs & MOUSE_BN1_DOWN) {
                printf("Got mouse click\n"
                    Post to the mouse semaphore
                DosPostEventSem(mouseClickSem);
            }
        }
}
        keyboardThread()
        KeyboardThread waits for a keystroke. When it gets one, it
    posts to keyboardSem.
void keyboardThread(void *foo)
    HKBD
                        keyboard;
    KBDKEYINFO
                        keyInfo;
        Get access to the keyboard
    KbdOpen(&keyboard);
    KbdGetFocus(IO_WAIT, keyboard);
    for (;;) {
        /*
* Wait for a keystroke
        KbdCharIn(&keyInfo, IO_WAIT, keyboard);
        printf("Got keystroke\n");
           Post to the semaphore
        DosPostEventSem(keystrokeSem);
}
```

Summary

Multiple wait semaphores, or muxwait semaphores, allow a thread to wait on several semaphores simultaneously. Using muxwait semaphores, a thread can wait on a group of event semaphores or a group of mutex semaphores. It can either wait for any one semaphore from the group, or it can wait for all of the semaphores in the group.

DosAddMuxWaitSem()

The DosAddMuxWaitSem() function allows an event or mutex semaphore to be added to the list of semaphores associated with a muxwait semaphore.

Prototype

#define INCL_DOSSEMAPHORES

#include <os2.h>

APIRET DosAddMuxWaitSem(HMUX hmux,

PSEMRECORD

pSemRec);

Parameters

hmux The handle of the muxwait semaphore.

pSemRec A pointer to the SEMRECORD that describes the event or the mutex

semaphore to add. A SEMRECORD has two parts: hsemCur (HSEM), which is a semaphore handle, and uluser (ULONG), which is a user ID.

Return value

- 0 NO_ERROR
- 6 ERROR INVALID HANDLE
- 8 ERROR NOT ENOUGH MEMORY
- 87 ERROR INVALID PARAMETER
- 100 ERROR_TOO_MANY_SEMAPHORES
- 105 ERROR SEM OWNER DIED
- 284 ERROR_DUPLICATE_HANDLE
- 292 ERROR_WRONG_TYPE

Description

The DosAddMuxWaitSem() allows a semaphore to be added to the list of semaphores associated with a muxwait semaphore. If the current list is made up of mutex semaphores, then only a mutex semaphore can be added. If the current list is made up of event semaphores, then only an event semaphore can be added.

DosCloseMuxWaitSem()

The DosCloseMuxWaitSem() releases a handle to a muxwait semaphore.

Prototype

Parameter

hmux The handle of the muxwait semaphore to close.

Return value

```
0 NO_ERROR
6 ERROR_INVALID_HANDLE
301 ERROR_SEM_BUSY
```

Description

The DosCloseMuxWaitSem() function releases a handle to a multiple wait semaphore. When all handles to a semaphore have been released, the semaphore itself is deleted.

DosCreateMuxWaitSem()

The DosCreateMuxWaitSem() function creates a multiple wait (muxwait) semaphore.

Prototype

APIRET DosCreateMuxWaitSem(
PSZ pszName,
PHMUX phmux,
ULONG cSemRec,
PSEMRECORD pSemRec,
ULONG flAttr);

Parameters

pszName A pointer to a null terminated string with the name of the semaphore

to create, or NULL to create an unnamed semaphore. A semaphore name

must begin with the prefix \SEM32\.

phmux A pointer to the location where the handle of the new semaphore will be

returned.

cSemRec The number of entries in semaphore list pointed to by pSemRec.

pSemRec A pointer to a list of semaphores.

flattr The creation attributes:

DC_SEM_SHARED Create semaphore shared.

DC MW WAIT ANY The semaphore triggers when any event semaphore

in psemRec is posted or when any mutex semaphore

in pSemRec is released.

DC MW WAIT ALL The semaphore triggers when all event semaphores

in pSemRec are posted or when all mutex sema-

phores in pSemRec are released.

Description

The DosCreateMuxWaitSem() function creates a multiple wait (muxwait) semaphore and returns its handle. A muxwait semaphore allows multiple event or mutex semaphores to be waited on with one DosWaitMuxWait() call. A list of component semaphores must be passed to the create call. These semaphores must either be all event semaphores or all mutex semaphores. Event and mutex semaphores cannot be mixed. The muxwait semaphore can be created to trigger either when any semaphore in the list triggers or when all semaphores in the list trigger.

The semaphore can be either named or unnamed depending on whether pszName specifies a string or is NULL. A semaphore can be made accessible by the creator or other processes via a call to DosOpenMuxWaitSem(), although it is not necessary

for the creator to explicitly open the semaphore. An unnamed semaphore can be created as shared by setting the DC_SEM_SHARED bit in the <code>flattr</code> parameter. Shared semaphores can be opened by processes other than the creator. Note that, because a shared, unnamed semaphore has no name, its handle must, in some way, be made available by the creator to processes that need to use it. All named semaphores are shared, so it is not necessary to explicitly set the DC_SEM_SHARED bit when creating a named semaphore.

DosDeleteMuxWaitSem()

The DosDeleteMuxWaitSem() function deletes an event or mutex semaphore from the list of semaphores associated with a muxwait semaphore.

Prototype

```
#define INCL_DOSSEMAPHORES tinclude <0e? h>
APIRET DosDeleteMuxWaitSem(
       HSEM
                      hSem);
```

Parameters

hmux The handle of the muxwait semaphore.

hSem The handle of the semaphore to remove from muxwait semaphore.

Return value

```
0 NO ERROR
 6 ERROR_INVALID_HANDLE
286 ERROR EMPTY MUXWAIT
```

Description

The DosDeleteMuxWaitSem() function deletes an event or mutex semaphore from the list of semaphores associated with a muxwait semaphore.

DosOpenMuxWaitSem()

The DosOpenMuxWaitSem() function opens a muxwait semaphore and returns a handle to it.

Prototype

#define INCL_DOSSEMAPHORES

Parameters

pszName A pointer to the null terminated string holding the name of the sema-

phore to open or NULL, if the semaphore handle is provided.

phmux A pointer to the location holding semaphore handle. If pszName is non-

NULL, phmux must be initialized to zero and the handle is returned. If pszName is NULL, phmux must be initialized to the handle of semaphore

to open.

Return value

- 0 NO ERROR
- 6 ERROR INVALID HANDLE
- 8 ERROR NOT ENOUGH MEMORY
- 87 ERROR INVALID PARAMETER
- 105 ERROR SEM OWNER DIED
- 123 ERROR_INVALID_NAME
- 187 ERROR SEM NOT FOUND
- 291 ERROR_TOO_MANY_OPENS

Description

The DosOpenMuxWaitSem() function opens a muxwait semaphore for use by a process. Named semaphores can be opened by name or handle; unnamed semaphores must be opened by handle.

DosQueryMuxWaitSem()

Function DosQueryMuxWaitSem() retrieves the semaphore list from a muxwait semaphore.

Prototype

INCL_DOSSEMAPHORES #define

#include <os2.h>

APIRET DosQueryMuxWaitSem(HMUX

pcSemRec, PULONG PSEMRECORD pSemRec, PULONG pflAttr);

Parameters

The muxwait semaphore handle. hmux

pcSemRec A pointer to the location that contains, on input, the size of the buffer

pointed to by pSemRec in PSEMRECORDs. The actual number of records

is returned.

A pointer to the buffer to hold semaphore records. pSemRec

A pointer to the location where the creation attributes are returned. pflAttr

> DC SEM SHARED Create semaphore shared.

DC MW WAIT ANY The semaphore triggers when any event semaphore

in pSemRec is posted or when any mutex sema-

phore in pSemRec is released.

DC MW WAIT ALL The semaphore triggers when all event semaphores

in pSemRec are posted or when all mutex sema-

phores in pSemRec are released.

Return value

0 NO ERROR

6 ERROR_INVALID_HANDLE

8 ERROR NOT ENOUGH MEMORY

87 ERROR INVALID PARAMETER

100 ERROR_TOO_MANY_SEMAPHORES

105 ERROR_SEM_OWNER_DIED

289 ERROR PARAM TOO SMALL

Description

Function DosQueryMuxWaitSem() retrieves the list of semaphore records associated with a muxwait semaphore. The list pointed to by pSemRec must be large enough to hold the number of semaphore record entries associated with the semaphore. If the buffer isn't large enough, the required size is returned in the location pointed to by pcSemRec and an error is returned.

DosWaitMuxWaitSem()

The DosWaitMuxWaitSem() function waits for a muxwait semaphore.

Prototype

#define INCL_DOSSEMAPHORES

 $\begin{array}{ll} \text{HMUX} & \textit{hmux}, \\ \text{ULONG} & \textit{ulTimeout}, \\ \text{PULONG} & \textit{pulUser}); \end{array}$

Parameters

hmux The muxwait semaphore handle.

ultimeout Sets timeout in milliseconds or 0 for SEM_IMMEDIATE_RETURN or -1

for SEM INDEFINITE WAIT.

pulUser A pointer to the location that receives the user ID for the semaphore

that was either posted or released.

Return value

0 NO ERROR

6 ERROR INVALID HANDLE

8 ERROR NOT ENOUGH MEMORY

87 ERROR INVALID PARAMETER

95 ERROR_INTERRUPT

103 ERROR_TOO_MANY_SEM_REQUESTS

105 ERROR SEM OWNER DIED

286 ERROR_EMPTY_MUXWAIT

287 ERROR MUTEX OWNED

292 ERROR_WRONG_TYPE

640 ERROR TIMEOUT

Description

The ${\tt DosWaitMuxWaitSem()}$ waits for the muxwait semaphore to reach the appropriate state.

If DC_MW_WAIT_ANY was specified in the creation of the semaphore, then this call will wait for any event semaphore in the muxwait list to become posted or any mutex semaphore in the muxwait list to become available. The user ID associated in the muxwait list with the semaphore posted or released will be returned in the location pointed to by pulUser.

If DC_MW_WAIT_ALL was specified, this call will wait for all event semaphores to become posted or all mutex semaphores to become available. The user ID associated in the muxwait list with the last semaphore posted or released will be returned in the

126 OS/2 Multithreading Facilities

location pointed to by pullser. Some care needs to be taken in using DC_MW_WAIT_ALL. If a muxwait semaphore is created to wait for all of a group of mutex semaphores to become available, no steps are taken to prevent starvation of the waiter. This means that, if one of the semaphores is always held by another thread, the muxwait will never trigger. In such cases, it is better to explicitly wait for each semaphore required. If a muxwait semaphore is created to wait for all of a group of event semaphores, the muxwait will not trigger until all of the event semaphores are posted at the same time. Thus, unlike single event waits, posting and immediately resetting event semaphores can prevent a muxwait from triggering.

Unless SEM_IMMEDIATE_RETURN is specified, this call will block until ownership can be passed to the current thread. If a timeout is specified, the call will return with an error if the timeout elapses before the semaphore is triggered.

Chapter

10

Using Queues

A queue allows orderly message passing between threads of one process or threads of different processes. Queues often are useful when threads have a client/server relationship. A *server* is any thread or process that provides services for other threads or processes. Client threads can send requests to a server that are executed on their behalf. For example, in chapter 14, we will see server threads that take requests to read and write files. The main thread will act as a client thread making requests of the servers.

Requests to a server thread can be composed and added to the server's request queue. Messages consist of three parts. First is a request ID. The request ID is simply a 32-bit number whose possible values are agreed upon by the client and the server. Typically, this is the type of request or command identifier. The second part is another 32-bit field that generally is the address of the message data. The third part is a 29-bit field that gives the length of the data.

Only the process that created the queue can read from it. Generally, a read from an empty queue will block, waiting for elements to be added to the queue. It is possible, however, to request that the read not block and that a semaphore be signalled when an element becomes available.

There are three types of queues. A FIFO queue orders the elements in first-infirst-out order. Thus, messages are removed from the queue in the same order that they are added. A LIFO queue orders the elements in last-in-first-out order. Here, the most recently added message is removed when the queue is read. The third queue type is a priority queue. Here, messages are ordered based on the priority associated with them when they are added to the queue. In a priority queue, messages of a given priority are processed in FIFO order.

While queues are largely intended for interprocess communication, they provide a very orderly and safe approach to interthread communication.

In program prog10.1.c, which is presented in Figure 10.1, we use a queue to transmit keystrokes from one thread to another. Here, we are passing a one-byte message with the ASCII code for the key. While a queue is overkill for this application, it is instructive to see how a simple task is performed with queues.

Figure 10.1 prog10-1.c

```
prog10-1.c
        This program demonstrates using queues.
        The main thread creates a queue and listens for requests.
    Requests consist of a keystroke. The second thread reads keys
    from the keyboard and adds requests to the queue one key at
    a time.
#include <stdio.h>
#include <stdlib.h>
#include <process.h>
#define INCL_DOSPROCESS
#define INCL DOSQUEUES
#define INCL_KBD
#include <os2.h>
#include "mt.h"
 * Function declarations for the threads.
void thread2(void *);
   Declare the queue handle
HQUEUE
                        queue;
        Main()
void main(int argc, char *argv[])
                         thread2Tid;
    REQUESTDATA
                         requestID;
    BYTE
                        priority;
    PVOID
                         buff;
    ULONG
                        buffLen:
        Create the queue
    DosCreateQueue(
                                             /* Queue handle
                         &queue,
                                             /* First In First Out
                         QUE FIFO.
                         "\\QUEUES\\FOOBAR");/* Queue name (mandatory)
        Start the second thread.
    thread2Tid= _beginthread(
                                             /* Address of function
                         thread2,
                                             /* Don't really give stack
                         NULL,
                                             /* Size of stack needed
                         STACK_SIZE,
                         (PVOID) 0);
                                             /* Message to thread
```

```
for (;;) {
         DosReadOueue(
                                                   /* Queue handle
                            queue,
                            &requestID,
                                                  /* User Request ID
                                                  /* Length of message
/* Address of message
/* Get first element
/* Wait for a request
/* Message priority
/* Sem handle (unused)
                           &buffLen.
                           &buff,
                           DCWW WAIT.
                           &priority,
                           NULL):
         /*
* Print a message
         printf("Received: %c (Pid #%d, request #%d)\n",
                           *((char *) buff),
                           requestID.pid,
                            requestID.ulData);
            Free the message
         free(buff);
         }
}
         thread2()
         Thread2 waits for keystrokes. On each keystroke, it sends a
    message on the queue with the ASCII value of the key.
void thread2(void *foo)
                           keyboard;
    HKRD
    KBDKEYINFO
                           keyInfo;
    ULONG
                           counter:
                          *key;
    char
    counter= 0;
     ^{\prime} * Get access to the keyboard. */
    KbdOpen(&keyboard);
    KbdGetFocus(IO_WAIT, keyboard);
    for (;;) {
        /*
* Wait for a keystroke
         KbdCharIn(&keyInfo, IO_WAIT, keyboard);
          * Add a request to the queue.
         key= malloc(1);
         *key= keyInfo.chChar;
         DosWriteQueue(
                                                   /* Queue handle
                                                                                    */
                       queue,
```

Figure 10.1 Continued

Program prog10-2.c, which is presented in Figure 10.2, performs the same task as prog10-1; however, it uses unpended queue reads. Notice that a semaphore now is required to synchronize the queue. When elements become available, the semaphore is signalled and the reading thread is awakened.

Figure 10.2 prog10-2.c

```
prog10-2.c
       This program demonstrates using queues with the NOWAIT
   option. Rather than pending on the queue read, we associate a
   semaphore with the queue and wait on that. Otherwise it
    functions just as prog10-1.c does.
#include <stdio.h>
#include <stdlib.h>
#include <process.h>
#define INCL_DOSPROCESS
#define INCL_DOSSEMAPHORES
#define INCL_DOSQUEUES
#define INCL_DOSERRORS
#define INCL_KBD
#include <os2.h>
#include "mt.h"
  Function declarations for the threads.
void thread2(void *);
 * Declare the queue handle
HQUEUE
                        queue;
       Main()
void main(int argc, char *argv[])
    TID
                        thread2Tid:
```

```
ULONG
                      err;
REQUESTDATA
                      requestID;
                      priority;
                      buff:
PVOTD
ULONG
                      buffLen:
HEV
                      queueSem;
ULONG
                      postCount;
/*

* Create the queue
DosCreateQueue(
                      &queue, /* Queue handle
OUE_FIFO, /* First In First Out
                       "\\QUEUES\\FOOBAR");/* Queue name (manditory)
* Create a semaphore to associate with the queue
DosCreateEventSem(
                      NULL,
                                             /* No name
                                            /* Semaphore handle
                      &queueSem,
                                            /* No options
                                             /* Initially unposted
                       FALSE):
* Start the second thread.
thread2Tid= _beginthread(
                                            /* Address of function
                      thread2,
                                            /* Don't really give stack */
                      NULL,
                                           /* Size of stack needed
                       STACK SIZE,
                                            /* Message to thread
                       (PVOID) 0):
for (;;) {
    err= DosReadQueue(
                      ue(
queue, /* Queue handle
&requestID, /* User Request ID
&bufflen, /* Length of message
&buff, /* Address of message
0, /* Get first element
DCWW_NOWAIT, /* DON'T wait for a request
&priority, /* Message priority

priority /* Sembandle
                      &priority,
                                            /* Sem handle
                      aueueSem);
    if (err == ERROR QUE EMPTY) {
        /*

* The queue was empty. Let's wait on the semaphore.
          * We'll wake up when something is added to the queue.
         DosWaitEventSem(queueSem, SEM_INDEFINITE_WAIT);
         DosResetEventSem(queueSem, &postCount);
          * It could conceivably have been posted more than once.
          * Get a queue message for each time.
         while (postCount") {
               * Get the message, we know it's there
              DosReadQueue(
                                           /* Queue handle
                                                                              */
                      queue,
```

Figure 10.2 Continued

```
/* User Request ID
                        &requestID,
                                            /* Length of message
                        &buffLen,
                                            /* Address of message
                        &buff,
                                            /* Get first element
                                            /* DON'T wait for a request
                        DCWW_NOWAIT,
                        &priority,
                                            /* Message priority
                                            /* Sem handle
                        queueSem);
                    Print it out
                printf(" Received: %c (pid %d, request %d)\n",
                            *((char *) buff),
                            requestID.pid,
                            requestID.ulData);
                    Free the message
                free(buff);
            }
        else {
                Print out the message. Mark it with a "*" to indicate
               that we didn't need to wait at all.
            printf("*Received: %c (pid %d, request %d)\n",
                            *((char *) buff),
                            requestID.pid,
                            requestID.ulData);
                Free the message
            free(buff);
        }
}
        thread2()
        Thread2 waits for keystrokes. On each keystroke, it sends a
    message on the queue with the ASCII value of the key.
void thread2(void *foo)
    HKBD
                        keyboard;
    KBDKEYINFO
                        keyInfo;
    ULONG
                        counter;
    char
                       *key;
    counter= 0;
        Get access to the keyboard.
```

```
KbdOpen(&kevboard):
     KbdGetFocus(IO WAIT, keyboard);
     for (;;) {
         /*

* Wait for a keystroke
         KbdCharIn(&keyInfo, IO_WAIT, keyboard);
             Add a request to the queue.
         key= malloc(1);
         *key= keyInfo.chChar;
         DosWriteQueue(
                                                    /* Queue handle
                       queue,
                                                   /* Request ID (made up)
/* Length of data
/* Address of data
/* Priority
                       counter++,
                       1,
                       key,
                       0);
         }
}
```

Summary

Queues provide a clean and simple way to pass messages from one thread to another. Often, queues are used when threads have a client/server relationship—in other words, when one thread provides services to another thread. However, queues are useful in any situation where messages are to be passed between threads.

DosCloseQueue()

The DosCloseQueue() function terminates access to a queue.

Prototype

Parameter

hq The handle of the queue to close.

Return value

```
0 NO_ERROR
337 ERROR_QUE_INVALID_HANDLE
```

Description

The DosCloseQueue() function terminates access to a queue for the requesting process. If the requesting process is the owner/creator if the queue, then the outstanding elements on the queue are deleted, the queue is deleted, and client threads will receive ERROR_QUE_INVALID_HANDLE errors on subsequent requests. If the requesting process is a client, access is terminated, but the queue itself is unaffected.

DosCreateQueue()

The DosCreateQueue() function creates a queue.

Prototype

Parameters

phq A pointer to the location where the new queue handle is returned.

priority The queue element priority ordering and flags:

- 0 QUE_FIFO
- 1 QUE_LIFO
- 2 QUE_PRIORITY
- 4 QUE_CONVERT_ADDRESS—the data pointer is converted from a 16-bit address to a 32-bit address.

pszName A pointer to the queue name (required).

Return value

```
0 NO_ERROR
87 ERROR_INVALID_PARAMETER
332 ERROR_QUEUE_DUPLICATE
334 ERROR_QUE_NO_MEMORY
135 ERROR_QUE_INVALID_NAME
```

Description

The DosCreateQueue() function creates a queue. A name must be specified. There are no unnamed queues. When creating a queue, the queue ordering must be specified. It can be either first-in-first-out, last-in-first-out, or priority based. The QUE_CONVERT_ADDRESS flag allows 16-bit processes to write to the queue. Their data buffer addresses are converted into 32-bit addresses.

DosOpenQueue()

The DosOpenQueue() function allows a client process to gain access to a queue.

Prototype

Parameters

ppid A pointer to the location where the queue owner's PID is returned.

phq A pointer to the location where the queue handle is returned.

pszName A pointer to the queue name.

Return value

```
0 NO_ERROR
334 ERROR_QUE_NO_MEMORY
341 ERROR_QUE_PROC_NO_ACCESS
343 ERROR_QUE_NAME_NOT_EXIST
```

Description

The DosOpenQueue() function allows processes other than the owner process to get access to a queue. Non-owner processes can only write to the queue; they cannot read, peek, or purge the queue.

DosPeekQueue()

The DosPeekQueue() function examines a data element in the queue without removing it from the queue.

Prototype

```
INCL_DOSQUEUES
#define
#include
              <os2.h>
APIRET DosPeekQueue(
     HQUEUE
                  hq,
      PREQUESTDATA pRequest,
      PULONG pcbData,
                  ppbuf,
      PPVOTD
                  element,
      PULONG
      BOOL32
                   nowait,
      PBYTE
                   ppriority,
      HEV
                   hsem);
```

Parameters

hg Holds the queue handle.

pRequest A pointer to the REQUESTDATA structure where the request ID

information is returned:

idpid PID of process that added the element to the queue.

ulData The request code that is specified by the application (there is no special meaning to this data).

pcbData Receives the length of the data element.

ppBuf Receives the address of the data element.

element A pointer to the location that indicates where in the queue to peek, on input:

O Examine first element of queue according to queue ordering

non-0 The element that was examined by the previous DosPeek Queue().

On output, <code>element</code> contains the identifier of the element that was peeked at. This can be used on a subsequent peek or read to target the same element.

nowait Holds no wait if queue is empty

- 0 DCWW_WAIT—the requesting thread waits for an element to be added to the queue.
- 1 DCWW_NOWAIT—the requesting thread does not wait for an element to be added to the queue.

ppriority A pointer to the location that receives the priority of element received as set by DosWriteQueue().

138 OS/2 Multithreading Facilities

hsem Holds the semaphore handle that is posted when data is added to the

queue

Return value

- 0 NO ERROR
- 87 ERROR INVALID PARAMETER
- 330 ERROR_QUE_PROC_NOT_OWNED
- 333 ERROR_ELEMENT_NOT_EXIST
- 337 ERROR_QUE_INVALID_HANDLE
- 342 ERROR QUE EMPTY
- 433 ERROR QUE INVALID WAIT

Description

The DosPeekQueue() function examines a message in the queue without removing it from the queue. Note that this function can be used only from within the owner process.

Normally, if the queue is empty, the peek will block waiting for an element to be added. However, if the DCWW_NOWAIT flag has been set, a semaphore must be provided that will be posted when an element is added. Subsequent calls to DosPeekQueue() must provide the same semaphore handle. Once the semaphore has been posted, a peek can be issued to examine the element.

DosPurgeQueue()

The DosPurgeQueue() function purges a queue. All the elements within the queue are unconditionally deleted.

Prototype

Parameter

hq Holds the queue handle.

Return value

```
0 NO_ERROR
330 ERROR_QUE_PROC_NOT_OWNED
337 ERROR_QUE_INVALID_HANDLE
```

Description

The DosPurgeQueue() function purges a queue. All elements in the queue are deleted. This function is available only to the queue owner.

DosReadQueue()

The DosReadQueue() function removes one element from a queue.

Prototype

INCL_DOSQUEUES #define #include <os2.h> APIRET DosReadQueue(HQUEUE hq, PREQUESTDATA pRequest, pcbData, PULONG PPVOID ppbuf, ULONG element, BOOL32 wait, ppriority, PBYTE HEV hsem);

Parameters

hg Holds the queue handle.

pRequest Receives the queue request ID. pRequest is a pointer to the two-double word data field.

Value Definition

- 1 ID of the process that added the element to the queue.
- The event code that is specified by the application (there is no special meaning to this data).

pcbData Receives the length of the data element.

Receives the address of the data element.

element Holds a particular element to request.

Remove the first element of the queue.

non-0 Remove the element that was examined by DosPeekQueue().

wait Holds no wait if the queue is empty.

- 0 DCWW_WAIT—the requesting thread waits for an element to be added to the queue.
- 1 DCWW_NOWAIT—the requesting thread does not wait for an element to be added to the queue.

priority Holds the priority of the element received as set by DosWrite Queue().

hsem Holds the semaphore handle that is posted when data is added to the queue.

Return value

- 0 NO ERROR
- 87 ERROR_INVALID_PARAMETER
- 330 ERROR_QUE_PROC_NOT_OWNED
- 333 ERROR_ELEMENT_NOT_EXIST
- 337 ERROR_QUE_INVALID_HANDLE
- 342 ERROR_QUE_EMPTY
- 433 ERROR_QUE_INVALID_WAIT

Description

The DosPeekQueue() function removes and returns one message from the queue. Note that this function can be used only from within the owner process.

Normally, if the queue is empty, the read will block waiting for an element to be added. However, if the DCWW_NOWAIT flag has been set, a semaphore must be provided that will be posted when an element is added. Subsequent calls to DosRead Queue() must provide the same semaphore handle. Once the semaphore has been posted, a read can be issued to retrieve the element.

DosQueryQueue()

The DosQueryQueue() function queries the number of elements currently in a queue.

Prototype

Parameters

```
hq Holds the queue handle.pcbEntries Points to the number of elements in the queue.
```

Return value

```
0 NO_ERROR
337 ERROR_QUE_INVALID_HANDLE
```

Description

The DosQueryQueue() function returns the number of elements currently in a queue. This function can be used by either server or client processes.

DosWriteQueue()

The DosWriteQueue() function adds a data element to a queue.

Prototype

Parameters

hq Holds the queue handle.

request The request or command code (application specific, the system does

not interpret this information).

cbData Holds the length of the data element.

pbData Holds the address of the data element.

priority Holds the priority of the element received as set by DosWrite

Queue ()—values can range from 0 to 15.

Return value

```
0 NO_ERROR
334 ERROR_QUE_NO_MEMORY
337 ERROR_QUE_INVALID_HANDLE
```

Description

The DosWriteQueue() function adds a data element to a queue. The element is positioned based on the ordering of the queue. If the queue is a priority queue, then a priority for the new element must be specified.

Chapter

Using Timers

Timers allow threads to be awakened based on the passage of time. There are two types of timers: single-interval and repeated-interval. A *single-interval timer* measures an interval of time. At the end of the interval, the timer posts a semaphore and disappears. A *repeated-interval timer* works similarly; however, after the interval elapses, it restarts so that the semaphore is posted periodically.

Single-interval timers are handy wherever an event needs to happen after some amount of time. Unlike DosSleep(), which pends the calling thread, a timer can be issued without pending the calling thread. A timer can be started by a thread other than the thread that will be waiting for the timer to elapse.

Repeated-interval timers can be used where there is processing that needs to happen periodically. For example, an auto-save thread could be awakened every five minutes to save work to disk.

Program prog11-1.c, which is presented in Figure 11.1, shows the basic operation of a single-interval timer. A five-second timer is started by thread 1. Thread 2 waits on the semaphore, then prints a message. Thread 1 prints messages every second. When the timer elapses, thread 2 is unblocked and prints its message.

Figure 11.1 progl1-1.c

prog11-1.c

This program demonstrates using asynchronous timers.

The main thread starts up another thread, then sets up a one-shot timer using DosAsynchTimer(). It then does some printing. The second thread waits on the semaphore associated with the timer. When the timer expires, the second thread is awakened and prints its message.

Figure 11.1 Continued

```
#include <stdio.h>
#include <stdlib.h>
#include <process.h>
#define INCL_DOSPROCESS
#define INCL_DOSSEMAPHORES
#define INCL_DOSDATETIME
#define INCL_KBD
#include <os2.h>
#include "mt.h"
/*

* Function declarations for the threads.
void thread2(void *);
    Define the semaphore
HEV
                         timerSem:
        Main()
void main(int argc, char *argv[])
    TID
                         thread2Tid;
    ULONG
                         i;
    HTIMER
                         timer;
    /*
 * Create the timer semaphore
 */
    DosCreateEventSem(
                         NULL,
                                               /* No name
                                              /* Semaphore handle
                         &timerSem.
                                              /* Timer requires Shared sem*/
                         DC_SEM_SHARED,
                                              /* Initially unposted
        Start the threads:
    thread2Tid= _beginthread(
                                              /* Address of function
                                              /* Don't really give stack
/* Size of stack needed
                         NULL,
                         STACK_SIZE,
                                              /* Message to thread
                         (PVOID) 0);
        Start the timer now
    DosAsyncTimer(
                         5000,
                                               /* Duration in msec
                                              /* Event semaphore
                         (HSEM) timerSem,
                                              /* Timer handle
                         &timer);
     * Do some stuff to look busy
*/
    for (i = 0; i < 15; i++) {
        printf("tick ");
```

```
fflush(stdout);
       DosSleep(500):
       printf("tock\n");
       DosSleep(500);
}
        thread2()
        thread2 waits on timerSem and prints its message when
    the semaphore is posted.
void thread2(void *thread2um)
   ULONG
                        postCount;
    for (;;) {
           Wait on the semaphore
       DosWaitEventSem(timerSem, SEM_INDEFINITE_WAIT);
           Reset it
       DosResetEventSem(timerSem, &postCount);
         * Print out our message
       printf(ANSI_RED " <BOOM!> \007\n" ANSI_RESET); fflush(stdout);
```

Program prog11.2-c, which is presented in Figure 11-2, does the same thing with a repeated-interval timer. Here, thread 2 is awakened every five seconds to print its message.

Figure 11.2 prog11-2.c

```
prog11-2.c

This program demonstrates using asynchronous timers. It uses DosStartTimer() to create a continuous timer.

The main thread starts up another thread, then sets up a repeating timer using DosStartTimer(). It then does some printing. The second thread waits on the semaphore associated with the timer. Each time the timer expires the second thread is awakened and prints its message.
```

Figure 11.2 Continued

```
#include <stdio.h>>
#include <stdlib.h>
#include <process.h>
#define INCL_DOSPROCESS
#define INCL_DOSSEMAPHORES
#define INCL_DOSDATETIME
#define INCL_KBD
#include <os2.h>
#include "mt.h"
/*
 * Function declarations for the threads.
 */
void thread2(void *);
   Define the semaphore
HEV
                         timerSem;
        Main()
void main(int argc, char *argv[])
    TID
                         thread2Tid;
    ULONG
                         i;
    HTIMER
                         timer;
     * Create the timer semaphore
    DosCreateEventSem(
                                              /* No name
                                              /* Semaphore handle
                         &timerSem,
                                              /* Timer requires Shared sem*/
                         DC_SEM_SHARED,
                                              /* Initially unposted
        Start the threads:
    thread2Tid= _beginthread(
                                              /* Address of function
                         thread2,
                                              /* Don't really give stack
                         NULL,
                         STACK_SIZE,
                                              /* Size of stack needed
                                              /* Message to thread
                         (PVOID) 0);
        Start the timer now
    DosStartTimer(
                         5000,
                                              /* Duration in msec
                                             /* Event semaphore
/* Timer handle
                         (HSEM) timerSem,
        Do some stuff to look busy
    for (i = 0; i < 15; i++) {
```

```
printf("tick ");
        fflush(stdout);
        DosSleep(500);
        printf("tock\n");
        DosSleep(500);
}
        thread2()
        Thread2 waits on timerSem and prints its message when
    the semaphore is posted.
void thread2(void *thread2um)
    ULONG
                        postCount;
    for (;;) {
            Wait on the semaphore
        DosWaitEventSem(timerSem, SEM INDEFINITE WAIT);
         * Reset it
        DosResetEventSem(timerSem, &postCount);
            Print out our message
        printf(ANSI_RED "<B00M!> \007\n" ANSI_RESET); fflush(stdout);
}
```

Summary

Timers allow a thread to get notified via a semaphore when an interval of time has elapsed. There are two types of timers: single-interval and repeated-interval. Single-interval timers trigger once, after the interval has elapsed. Repeated-interval timers are reset each time the interval elapses, thus triggering periodically.

DosAsyncTimer()

The DosAsyncTimer () function starts the operation of an asynchronous timer.

Prototype

Parameters

msec Holds the time interval in milliseconds before the event semaphore spec-

ified by *hsem* is posted.

hsem Holds the event semaphore handle that will be posted when the msec

time interval has elapsed.

phtimer A pointer to the location in which the timer handle is returned.

Return value

0 NO_ERROR

323 ERROR TS SEMHANDLE

324 ERROR_TS_NOTTIMER

Description

The DosAsyncTimer() function starts "one-shot" timer. The timer runs asynchronously to the calling thread and posts the specified event semaphore when the designated time interval is reached. The time interval is specified in milliseconds. Note that the specified time interval must be considered as approximate because the actual duration of the elapsed time will be affected by system operations factors.

DosStartTimer()

The DosStartTimer() function starts the operation of an asynchronous repeated-interval timer.

Prototype

Parameters

msec Holds the time interval in milliseconds before the event semaphore spec-

ified by hsem is posted.

hsem Holds the event semaphore handle that will be posted when the msec

timer interval has been reached.

phtimer A pointer to the location in which the timer handle is returned.

Return value

0 NO_ERROR 323 ERROR_TS_SEMHANDLE 324 ERROR TS NOTTIMER

Description

The DosStartTimer() function starts an asynchronous repeated-interval timer. The timer runs asynchronously to the calling thread and posts an event semaphore every time the designated time interval has elapsed. The time interval is specified in milliseconds. Note that the specified time interval must be considered as approximate because the actual duration of the elapsed time will be affected by system operations factors.

DosStopTimer()

The DosStopTimer() function stops the operation of an asynchronous repeated-interval timer.

Prototype

```
#define INCL_DOSDATETIME
#include <os2.h>

APIRET DosStopTimer(
HTIMER htimer);
```

Parameter

htimer The handle of the timer to be stopped.

Return value

```
0 NO_ERROR
326 ERROR_TS_HANDLE
```

Description

The DosStopTimer() function stops the operation of either an asynchronous repeated-interval timer (initiated by DosStartTimer()) or a single-interval timer (initiated by DosAsynchTimer()). Note that when DosStopTimer() is used no assumption about the state of the event semaphore can be made. This means that you must call DosResetEventSem() before you restart the timer.

Real Multithreading Programs

Chapter

12

A Simple Problem, Many Incorrect Solutions

This chapter begins our introduction to writing multithreaded programs. Writing multithreaded programs is hard. More specifically writing correct multithreaded programs is hard. The purpose of this chapter is to instill in you a certain respect for the power and subtlety of multithreading. You will see just how easy it is to believe that an incorrect program is correct.

This chapter started as a simple sample program. As we developed a bulletproof solution, we found that the problem was much more subtle and difficult than we had at first imagined it would be. We will take you through a hypothetical development process, starting with a clearly inadequate solution and building up to a correct solution.

Our program is to have five threads, the main thread and four auxiliary threads. The auxiliary thread have some processing that they do repeatedly. The goals of a solution are as follows:

- The main thread must be able to start and stop the auxiliary threads.
- The auxiliary threads must stop at a known location in their processing loops.
- The main thread must know when all of the auxiliary threads have reached their stopping point.
- The program must exhibit liveness, in other words, it must not hang.

Program prog12-1.c, which is presented in Figure 12.1, is a first draft of a solution. The auxiliary threads run a loop that looks like this:

```
loop
  wait on goSem
    .. processing ..
endloop
```

Figure 12.1 prog12-1.c

```
prog12-1.c
        This program is the first incorrect solution to the problem
   of stopping and starting several threads.
        The program creates four threads that are to be stopped
    and started by the main thread. We would like for the main
    thread to know when they have stopped. In this version, no
    attempt is made to synchronize the main thread.
#include <stdio.h>
#include <stdlib.h>
#include <process.h>
#define INCL_DOSPROCESS
#define INCL DOSSEMAPHORES
#include <os2.h>
#include "mt.h"
   Function declarations for the threads.
void threadN(void *);
/*
* Declare our semaphores
HEV
                        goSem;
 * Declare a global that will indicate the stage we're at
long
                        stage;
        Main()
void main(int argc, char *argv[])
    TID
                        thread[4];
    ULONG
    ULONG
                        eventCount;
       Create the event semaphore
    DosCreateEventSem(
                                             /* Local semaphore, no name */
                        NULL.
                                             /* Addr of semaphore handle */
                        &goSem,
                                             /* No options
                        0,
                                             /* Initially not posted
                        FALSE);
        Start the other threads
    for (i = 0; i < 4; i++) {
        thread[i]= beginthread(
```

```
/* Address of function
                        threadN,
                                             /* Don't really give stack
                        NULL,
                                             /* Size of stack needed
                        STACK_SIZE,
                        (PVOID) i);
                                             /* Message to thread
        }
    for (stage= 1; stage <= 5; stage++) {
        /*
* Post to the semaphore allowing the tasks to run.
        DosPostEventSem(goSem);
         * Let them run for a little bit
        DosSleep(400);
           And stop any movement.
        DosResetEventSem(goSem, &eventCount);
         * Utter what is likely to be a lie.
        printf("All threads are stopped at stage %d\n", stage);
            Wait a while
        DosSleep(2000);
        Print final message
    printf("We're done.\n");
}
        threadN()
        ThreadN prints messages. It uses handshaking with the main
    thread to stop and start.
void threadN(void *threadNum)
    USHORT
                        me;
   me= (USHORT) threadNum;
    for (;;) {
         ^{\star} Do the Wait. If the main task's reset goSem, then I'll pend.
        DosWaitEventSem(goSem, SEM_INDEFINITE_WAIT);
        printf("Thread %d crosses event at stage %d\n", me, stage);
        }
}
```

The main thread starts the auxiliary threads by resetting the gosem semaphore and stops them by resetting the semaphore. It is easy to see, based on the behavior of semaphores, that this solution satisfies conditions 1, 2, and 4. When the semaphore is reset, the threads ultimately will block on gosem, they will block at a known location in their loop, and the program will not hang.

Unfortunately, condition 3 is not satisfied. No attempt has been made to satisfy it. After resetting the semaphore, the main thread makes no attempt to determine when the auxiliary threads have stopped. An auxiliary thread might have just passed the wait on gosem when the main thread resets it. Thus, the auxiliary threads might get in one loop after gosem has been reset. The problematic sequence is as follows:

```
main thread

auxiliary thread #N

wait on goSem (no block)

reset goSem

thinks aux is stopped

processing...

wait on goSem (block)

continue
```

In prog12-1, as in the other examples in this chapter, we have used DosSleep() calls to open up scheduling windows. In this case, the scheduling window is between the DosWaitEventSem() and the printf(), so we insert a DosSleep() there to ensure that the race condition occurs.

A correct multithreaded program should behave correctly even if DosSleep()'s are inserted in the code. The insertion of a DosSleep() call does nothing that can't happen theoretically under normal processing. Control can be taken from a thread at any point and might not be returned for an arbitrarily long time. Thus, we can use DosSleep() to induce reschedules at certain points to illustrate errors in a program.

If you run prog12-1, you will find that, after the main thread has declared that the auxiliary threads have stopped, they still get one loop in. We addressed three out of four of our conditions without trying hard, so let's put a little effort in and see if we can satisfy all of our conditions.

The condition that we failed to satisfy in prog12-1 was that of letting the main thread know when the other threads had stopped. What's necessary is some communication from the auxiliary threads to the main thread.

In prog12-2.c, which is presented in Figure 12.2, we add an array of event semaphores—one for each auxiliary thread and a muxwait semaphore that triggers when all of these event semaphores have been posted. The event semaphores are posted by the auxiliary threads when they are about to wait on gosem. The new auxiliary thread loop looks like this:

```
loop
  post threadWaitingSem[my thread number]
  wait on goSem
    .. processing ..
endloop
```

The main thread still posts goSem to start the auxiliary threads. To stop them, however, it now executes the following code:

```
for i from 1 to 4
  reset threadWaitingSem[i]
  endfor
reset goSem
muxwait on threadWaitingSem[1-4]
```

Figure 12.2 prog12-2.c

```
prog12-2.c
        This program is the second incorrect solution to the problem
    of stopping and starting several threads.
        The program creates four threads that are to be stopped
    and started by the main thread. We would like for the main
    thread to know when they have stopped. In this version, we use
    four semaphores and a muxwait semaphore comprised of those
    four semaphores as a signal to the main thread that the
    auxiliary threads have stopped. This version is not correct,
    and we insert a DosSleep() to open up a window which illustrates
    the failure.
#include <stdio.h>
#include <stdlib.h>
#include <process.h>
#define INCL_DOSPROCESS
#define INCL DOSSEMAPHORES
#include <os2.h>
#include "mt.h"
    Function declarations for the threads.
void threadN(void *);
   Declare our semaphores
HEV
                        goSem;
HEV
                        threadWaitingSem[4];
HMUX
                        allThreadsWaitingSem;
char
                        synchRequested= FALSE;
   Declare a global that will indicate the stage we're at
long
                        stage;
        Main()
void main(int argc, char *argv[])
    TID
                        thread[4];
    ULONG
    ULONG
                        eventCount;
    ULONG
                        lastEvent;
    SEMRECORD
                        semaphoreList[4];
       Create all of the event semaphores
```

Figure 12.2 Continued

```
DosCreateEventSem(
                                       /* Local semaphore, no name */
                   NULL.
                   &goSem,
                                       /* Addr of semaphore handle */
                                       /* No options
                   FALSE):
                                       /* Initially not posted
for (i = 0; i < 4; i++) {
   DosCreateEventSem(
                                      /* Local semaphore, no name */
                   FALSE):
                                       /* Initially not posted
       Fill in the semaphore list for the MuxWait
    semaphoreList[i].hsemCur=\ (HSEM)\ threadWaitingSem[i];
   Create the MuxWait semaphore
DosCreateMuxWaitSem(
                   NULL.
                                       /* Local semaphore, no name */
                   &allThreadsWaitingSem,/* Addr of semaphore handle */
                                     /* Three mutex semaphores
                                      /* Array of semaphores
                   semaphoreList,
                                      /* Wait till all are free
                   DCMW_WAIT_ALL);
   Start the other threads
for (i = 0; i < 4; i++) {
    thread[i]= _beginthread(
                                      /* Address of function
                   threadN,
                                      /* Don't really give stack
                   NULL,
                                      /* Size of stack needed
                   STACK SIZE,
                                      /* Message to thread
                   (PVOID) i):
    }
for (stage= 1; stage <= 5; stage++) {
       Post to the semaphore allowing the tasks to run.
    DosPostEventSem(goSem);
       Let them run for a little bit
    DosSleep(400);
        And stop any movement.
        We reset the threadWaitingSem semaphores and the
        goSem semaphore. We want signalling of the threadWaitingSem
        semaphores to tell us that the threads are about to pend.
        There is a window, however, between the resets. We reset
        the threadWaitingSem's before we reset goSem. Some of the
        threads might signal their threadWaitingSem's and continue before
```

```
* we reset goSem. Thus, we won't wait for these threads.
         */
        for (i = 0; i < 4; i++) {
           DosResetEventSem(threadWaitingSem[i], &eventCount);
/**/
                                                        */
        DosSleep(1000);
                                /* Open up window
        DosResetEventSem(goSem, &eventCount);
           Wait for all movement to cease. When we come back,
           we will know (think) that all of the tasks passed the point
          of posting to the threadWaitingSem while the goSem was
         * reset. Thus, we know (think) that they are all waiting, or
           just about to wait.
        DosWaitMuxWaitSem(
                                                /* Semaphore handle
                        allThreadsWaitingSem,
                                                /* Wait forever
                        SEM_INDEFINITE_WAIT,
                        &lastEvent):
                                                /* Unused
        printf("All threads are stopped at stage %d\n", stage);
           Wait awhile
       DosSleep(2000);
   /*
* Print final message
   printf("We're done.\n");
        threadN()
        ThreadN prints messages. It uses handshaking with the main
    thread to stop and start.
void threadN(void *threadNum)
   lona
                       me:
   me= (long) threadNum;
   for (;;) {
           Post to my threadWaitingSem[]. This tells the main task
           that I'm about to do my Wait.
       DosPostEventSem(threadWaitingSem[me]);
           Do the Wait. If the main task's reset goSem, then I'll pend.
        * It also will have reset my threadWaitingSem, so my post will
        * have told it that I'm here.
```

Figure 12.2 Continued

The theory is this: after having reset the threadWaitingSem[]'s and goSem, when the main thread returns from the muxwait, all of the threadWaitingSem[]'s will have been posted since goSem was reset. Thus, we know that all of the auxiliary threads are waiting on goSem or are about to wait on goSem. This theory is false for the following reason: the auxiliary threads can run in between the time that we reset its threadWaitingSem[] and the time that we reset goSem. This allows the following sequence to occur:

This problem is fixed easily enough. We need to make the semaphore resetting atomic with respect to the other threads. So, in prog12-3.c, which is presented in Figure 12.3, we have surrounded the semaphore resets with DosEnterCritSec() and DosExitCritSec(). These calls prevent the auxiliary threads from running in between the semaphore reset calls. The main thread code now looks like this:

```
DosEnterCritSec()
for i from 1 to 4
  reset threadWaitingSem[i]
  endfor
reset goSem
DosExitCritSec()
muxwait on threadWaitingSem[1-4]
```

The call to DosEnterCritSec() ensures that the main thread is the only thread to run until the DosExitCritSec(). Thus, the previous race condition is avoided. When we return from the muxwait, we know that all of the threadWaitingSem[]'s were posted since we reset goSem. So, we know that the auxiliary threads are waiting on goSem or are about to wait on goSem. Everything looks good. However, this is only the third example, so it must have a problem. Indeed, it does.

One of the problems in multithreading is balancing synchronization against liveness. If there is not enough synchronization between threads, threads step on each other. If there is too much synchronization, you might have introduced a deadlock. In this program, we have introduced a deadlock. Consider the following sequence of events:

Figure 12.3 prog12-3.c

```
prog12-3.c
        This program is the third incorrect solution to the problem
    of stopping and starting several threads.
        The program creates four threads that are to be stopped
    and started by the main thread. We would like for the main
    thread to know when they have stopped. In this version, we
    correct the error of prog12-2 only to find another race
    condition.
#include <stdio.h>>
#include <stdlib.h>
#include <process.h>
#define INCL DOSPROCESS
#define INCL_DOSSEMAPHORES
#include <os2.h>
#include "mt.h"
/*

* Function declarations for the threads.
void threadN(void *);
* Declare our semaphores
HEV
                        goSem;
HEV
                        threadWaitingSem[4];
HMUX
                        allThreadsWaitingSem;
                        synchRequested= FALSE;
char
   Declare a global which will indicate the stage we're at
long
                        stage;
       Main()
void main(int argc, char *argv[])
                        thread[4];
   ULONG
                        i;
   ULONG
                        eventCount;
   ULONG
                        lastEvent;
   SEMRECORD
                        semaphoreList[4];
       Create all of the event semaphores
   DosCreateEventSem(
                        NULL.
                                            /* Local semaphore, no name */
                                            /* Addr of semaphore handle */
                        &goSem,
```

Figure 12.3 Continued

```
/* No options
                                       /* Initially not posted
for (i = 0; i < 4; i++) {
   DosCreateEventSem(
                                        /* Local semaphore, no name */
                    &threadWaitingSem[i],/* Addr of semaphore handle */
                                       /* No options
                                        /* Initially not posted
                    FALSE):
    ^{\prime} ^{\star} Fill in the semaphore list for the MuxWait
    semaphoreList[i].hsemCur= (HSEM) threadWaitingSem[i];
   Create the MuxWait semaphore
DosCreateMuxWaitSem(
                    NULL,
                                        /* Local semaphore, no name */
                    &allThreadsWaitingSem,/* Addr of semaphore handle */
                    4, /* Three mutex semaphores */
                                       /* Array of semaphores
                    semaphoreList,
                    DCMW_WAIT_ALL);
                                       /* Wait till all are free */
/*

* Start the other threads
for (i = 0; i < 4; i++) {
   thread[i]= _beginthread(
                    threadN,
                                       /* Address of function
                                       /* Don't really give stack */
                    NULL,
                    STACK_SIZE,
                                       /* Size of stack needed
                                       /* Message to thread
                    (PVOID) i);
    }
for (stage= 1; stage <= 5; stage++) {
    ^{\prime*} ^{} Post to the semaphore allowing the tasks to run.
    DosPostEventSem(goSem);
    * Let them run for a little bit
    DosSleep(400);
     * And stop any movement.
     ^{\star} We enclose all of the resets in a critical section. This
       ensures that when we wake up below, threadWaitingSem[i] will
       have been signalled while goSem was reset, ensuring that
       the threads are pended or about to pend.
    DosEnterCritSec();
    for (i = 0; i < 4; i++) {
```

0

```
DosResetEventSem(threadWaitingSem[i], &eventCount);
        DosResetEventSem(goSem, &eventCount);
        DosExitCritSec();
           Wait for all movement to cease. When we come back,
           we will know that all of the tasks passed the point
           of posting to the threadWaitingSem while the goSem was
         * reset. Thus, we know that they are all waiting, or just about
        * to wait.
        DosWaitMuxWaitSem(
                       allThreadsWaitingSem,
                                                /* Semaphore handle
                                                /* Wait forever
                       SEM_INDEFINITE_WAIT,
                                                /* Unused
                       &lastEvent);
        printf("All threads are stopped at stage %d\n", stage);
        * Wait a while
       DosSleep(2000);
       Print final message
   printf("We're done.\n");
        threadN()
        ThreadN prints messages. It uses handshaking with the main
    thread to stop and start.
void threadN(void *threadNum)
{
    long
                       me;
   me= (long) threadNum;
   for (;;) {
       /*
* Post to my threadWaitingSem[]. This tells the main task
        * that I'm about to do my Wait.
       DosPostEventSem(threadWaitingSem[me]);
        * The problem here is that, if the main thread resets
        * all of the semaphores here, we are past the point
        * of signalling, but we will pend on the wait, never
        * signalling the main thread and causing a deadlock.
```

Figure 12.3 Continued

```
main thread auxiliary thread #N
post threadWaiting[N]
enter critSec
reset threadWaiting[1]
reset threadWaiting[2]
reset threadWaiting[3]
reset threadWaiting[4]
reset goSem
exit critSec
wait on goSem (block)
```

So, we have yet another race condition. This one causes a hang because the main thread resets all of the semaphores after the auxiliary thread has posted its thread WaitingSem[] but before it gets to the wait on goSem. So, it waits on goSem, not having posted threadWaitingSem[] since the reset, and the main thread waits for threadWaitingSem[].

How to fix this? The last time that we had a window, between the threadWaitingSem[] resets and the goSem reset, we plugged it up by surrounding the section with a DosEnterCritSec()-DosExitCritSec() pair. Note that we cannot do this here. The reason is that part of the critical section would be a call that blocks. This would cause a deadlock. The auxiliary thread would disable rescheduling, preventing the main thread from running, then wait on goSem until the main thread posted it. We'd be assured of a deadlock.

We need to find a more subtle solution. The problem that we had in the previous program was that the auxiliary threads went through the post and wait on every loop. If we could avoid this and execute this code only when we wanted to, the main thread would know that the auxiliary threads were not in that code path, eliminating the window. We do this in prog12-4.c, which is presented in Figure 12.4, by adding a variable synchRequested. The auxiliary threads go into the synchronization code only when this flag is set. The main thread sets the flag when it is in the critical section. It resets the flag after all of the auxiliary threads have stopped. The code for the auxiliary threads looks like this:

```
loop
  if synchRequested
     post threadWaiting[my thread number]
     wait on goSem
```

```
endif
.. processing..
endloop
```

The main thread stops the threads with the following code:

```
DosEnterCritSec()
synchRequested= FALSE;
for i from 1 to 4
   reset threadWaitingSem[i]
   endfor
   reset goSem
   DosExitCritSec()
   muxwait on threadWaitingSem[1-4]
synchRequested= FALSE;
```

Figure 12.4 prog12-4.c

```
* prog12-4.c

This program is the first incorrect solution to the problem of stopping and starting several threads.

The program creates four threads that are to be stopped and started by the main thread. We would like for the main thread to know when they have stopped. In this version, we fix the problem with prog12-3.c by introducing a communication flag. The auxiliary threads now post threadWaitingSem and wait on goSem only if synchRequested is TRUE. This solves some problems and actually makes the auxiliary threads faster, because they only do the calls when the main thread wants to block them.
```

```
#include <stdio.h>
#include <stdlib.h>
#include <process.h>
#define INCL_DOSPROCESS
#define INCL DOSSEMAPHORES
#include <os2.h>
#include "mt.h"
 * Function declarations for the threads.
void threadN(void *);
 * Declare our semaphores
HEV
                        goSem;
HEV
                        threadWaitingSem[4];
HMUX
                        allThreadsWaitingSem;
char
                        synchRequested= FALSE;
 * Declare a global that will indicate the stage we're at
```

Figure 12.4 Continued

```
long
                        stage;
        Main()
void main(int argc, char *argv[])
    TID
                        thread[4];
    ULONG
    ULONG
                        eventCount;
    ULONG
                        lastEvent:
    SEMRECORD
                        semaphoreList[4];
        Create all of the event semaphores
    DosCreateEventSem(
                        NULL.
                                             /* Local semaphore, no name */
                        &goSem,
                                             /* Addr of semaphore handle */
                                             /* No options
                                            /* Initially not posted
                        FALSE);
    for (i = 0; i < 4; i++) {
        DosCreateEventSem(
                                             /* Local semaphore, no name */
                        &threadWaitingSem[i],/* Addr of semaphore handle */
                                            /* No options
                                             /* Initially not posted
                        FALSE);
            Fill in the semaphore list for the MuxWait
        semaphoreList[i].hsemCur= (HSEM) threadWaitingSem[i];
        Create the MuxWait semaphore
    DosCreateMuxWaitSem(
                        NULL.
                                             /* Local semaphore, no name */
                        &allThreadsWaitingSem,/* Addr of semaphore handle */
                                            /* Three mutex semaphores
                                             /* Array of semaphores
                        semaphoreList,
                        DCMW_WAIT_ALL);
                                             /* Wait till all are free
        Start the other threads
    for (i = 0; i < 4; i++) {
        thread[i]= _beginthread(
                                             /* Address of function
                        threadN,
                                             /* Don't really give stack
                        NULL,
                        STACK_SIZE,
                                             /* Size of stack needed
                                             /* Message to thread
                        (PVOID) i);
        }
    for (stage= 1; stage <= 5; stage++) {
```

```
/*
* Post to the semaphore allowing the tasks to run.
    DosPostEventSem(goSem);
    /*
 * Let them run for a little bit
 */
    DosSleep(400);
     * And stop any movement.
     ^{\star} We enclose all of the resets in a critical section. This
        ensures that when we wake up below, threadWaitingSem[i] will
     * have been signalled while goSem was reset, ensuring that
     * the threads are pended or about to pend.
    DosEnterCritSec();
    /*
* Because the auxiliary threads now do synchronization
** TRUE we set it now.
     * only when synchRequested is TRUE, we set it now.
    synchRequested= TRUE:
    for (i = 0; i < 4; i++) {
        DosResetEventSem(threadWaitingSem[i], &eventCount);
    DosResetEventSem(goSem, &eventCount);
    DosExitCritSec();
    /*
* Wait for all movement to cease. When we come back,
     ^{\star} we will know that all of the tasks passed the point
     ^{\star} of posting to the threadWaitingSem while the goSem was
     * reset. Thus, we know that they are all waiting, or just about
     * to wait.
    DosWaitMuxWaitSem(
                     allThreadsWaitingSem,
                                              /* Semaphore handle
                                              /* Wait forever
                     SEM_INDEFINITE_WAIT,
                     &lastEvent);
                                              /* Unused
     * They're all waiting. It's safe to set synchRequested to
    * FALSE.
     */
    synchRequested= FALSE;
    printf("All threads are stopped at stage %d\n", stage);
       Wait a while
    DosSleep(2000);
/*
```

Figure 12.4 Continued

```
Print final message
    printf("We're done.\n");
}
        threadN()
        ThreadN prints messages. It uses handshaking with the main
    thread to stop and start.
void threadN(void *threadNum)
    long
                        me;
   me= (long) threadNum;
    for (;;) {
           Post to my threadWaitingSem[]. This tells the main task
           that I'm about to do my Wait.
        if (synchRequested) {
           DosPostEventSem(threadWaitingSem[me]);
               Here, we have a window in the same place; however,
               the problem is different. The problem now is if
               the main thread attempts to block us, but we get rescheduled
              here long enough for the main thread to unblock us, and
               block us again.
             * Like prog12-3, we will hang because we will be waiting
              for the main thread to post goSem, and the main thread will
               be waiting for us to post threadWaitingSem[i].
    /**/
           DosSleep(10000);
                                     /* Open up window */
               Do the Wait. If the main task's reset goSem, then I'll pend.
               It also will have reset my threadWaitingSem, so my post will
             * have told it that I'm here.
            DosWaitEventSem(goSem, SEM_INDEFINITE_WAIT);
        printf("Thread %d crosses event at stage %d\n", me, stage);
```

One significant advantage of this program is that less time is spent by the auxiliary threads doing synchronization because they do not make post and wait calls in the general case, only when synchronization is requested. Because the auxiliary threads go into the troublesome area only when the main thread tells them to, everything's OK, right? Of course not. We have yet another race condition. This one is rather subtle. This code works just fine once. To see this race condition, you need to consider

that the main thread is stopping, starting, stopping, etc., the auxiliary threads. The window in the auxiliary thread is the same. However, the code that the main thread might execute that causes a problem is different. Consider the following sequence:

```
auxiliary thread #N
main thread
                            .. processing ..
DosEnterCritSec()
synchRequested= FALSE;
reset threadWaitingSem[1]
reset threadWaitingSem[2]
reset threadWaitingSem[3]
reset threadWaitingSem[4]
reset goSem
DosExitCritSec()
muxwait (blocks)
                            if (synchRequested) (true)
                            post threadWaitingSem[N]
synchRequested= FALSE;
print("threads stopped")
DosSleep()
-top of loop-
post goSem
DosSleep()
DosEnterCritSec()
synchRequested= FALSE;
reset threadWaitingSem[1]
reset threadWaitingSem[2]
reset threadWaitingSem[3]
reset threadWaitingSem[4]
reset goSem
DosExitCritSec()
muxwait (blocks)
                            wait on goSem (blocks)
```

It's easy to see that we still have a problem. We need to break down and add some more semaphores into the mix. In prog12-5.c, which is presented in Figure 12.5, we add another semaphore per auxiliary threads and another muxwait semaphore. These semaphores are used by the auxiliary threads to tell the main thread that they are outside of the synchronization processing. When the main thread wants to stop the auxiliary threads, it must first wait until they are out of any previous synchronization processing they might be doing. The new auxiliary thread code looks like this:

```
loop
  if (synchRequested)
    reset threadRunningSem[my thread num]
    post threadWaitingSem[my thread num]
    wait on goSem
    post threadRunningSem[my thread num]
    endif
    . processing ..
endloop
```

For a given auxiliary thread, threadRunningSem[] is in the posted state only when the thread is outside of the synchronization code. This allows the main thread to wait for all of the auxiliary threads to get running. The main thread now looks like this:

```
muxwait on threadRunningSem[1-4]
DosEnterCritSec()
```

```
synchRequested= FALSE;
for i from 1 to 4
  reset threadWaitingSem[i]
  endfor
reset goSem
DosExitCritSec()
muxwait on threadWaitingSem[1-4]
synchRequested= FALSE;
```

Note that, rather than waiting for the auxiliary threads to get running after we post goSem at the top of the main thread loop, we wait for them to get running just before we stop them. This is the first time that we actually need them to get out of the if-statement. It is more efficient to wait later rather than earlier because they most likely will all have started to run by the time that we do the first muxwait. This way we'll rarely have to wait.

Figure 12.5 prog12-.5.c

```
#include <stdio.h>
#include <stdlib.h>
#include cess.h>
#define INCL_DOSPROCESS
#define INCL DOSSEMAPHORES
#include <os2.h>
#include "mt.h"
   Function declarations for the threads.
void threadN(void *);
* Declare our semaphores
HEV
                        goSem;
HEV
                        threadWaitingSem[4];
                        threadRunningSem[4];
HEV
HMUX
                        allThreadsWaitingSem;
HMUX
                        allThreadsRunningSem;
char
                        synchRequested= FALSE;
```

```
* Declare a global that will indicate the stage we're at
long
                        stage;
       Main()
void main(int argc, char *argv[])
   TID
                         thread[4];
   ULONG
                        i;
   ULONG
                        eventCount;
   UL ONG
                        lastEvent;
   SEMRECORD
                        semaphoreList[4];
       Create all of the event semaphores
   DosCreateEventSem(
                        NULL.
                                             /* Local semaphore, no name */
                                             /* Addr of semaphore handle */
                        &goSem,
                        0,
                                             /* No options
                        FALSE);
                                             /* Initially not posted
   for (i = 0; i < 4; i++) {
       DosCreateEventSem(
                                             /* Local semaphore, no name */
                        &threadWaitingSem[i],/* Addr of semaphore handle */
0, /* No options */
                                             /* Initially not posted
                        FALSE);
            Fill in the semaphore list for the MuxWait
        semaphoreList[i].hsemCur= (HSEM) threadWaitingSem[i];
       Create the MuxWait semaphore
   DosCreateMuxWaitSem(
                                             /* Local semaphore, no name */
                        NULL.
                        &allThreadsWaitingSem,/* Addr of semaphore handle */
                                             /* Three mutex semaphores
                                             /* Array of semaphores
                        semaphoreList,
                                             /* Wait till all are free
                        DCMW_WAIT_ALL);
   for (i = 0; i < 4; i++) {
        DosCreateEventSem(
                                             /* Local semaphore, no name */
                        &threadRunningSem[i],/* Addr of semaphore handle */
                                             /* No options
                                             /* Initially posted
                        TRUE ):
            Fill in the semaphore list for the MuxWait
        semaphoreList[i].hsemCur= (HSEM) threadRunningSem[i];
       Create the MuxWait semaphore
```

{

Figure 12.5 Continued

```
DosCreateMuxWaitSem(
                                        /* Local semaphore, no name */
                    &allThreadsRunningSem,/* Addr of semaphore handle */
                                       /* Three mutex semaphores */
                                       /* Array of semaphores
                    semaphoreList,
                                        /* Wait till all are free */
                    DCMW_WAIT_ALL);
   Start the other threads
for (i = 0; i < 4; i++) {
    thread[i]= _beginthread(
                                        /* Address of function
                    threadN,
                                        /* Don't really give stack
                    NULL,
                                        /* Size of stack needed
                    STACK_SIZE,
                                        /* Message to thread
                    (PVOID) i);
    }
for (stage= 1; stage <= 5; stage++) {
    ^{\prime} * Post to the semaphore allowing the tasks to run.
   DosPostEventSem(goSem);
   /*

* Let them run for a little bit
   DosSleep(400);
    ^\prime ^\star Wait for them to get out of the synchronization block
   DosWaitMuxWaitSem(
                    allThreadsRunningSem,
                    SEM_INDEFINITE_WAIT,
                    &lastEvent);
       And stop any movement.
     * We enclose all of the resets in a critical section. This
       ensures that when we wake up below, threadWaitingSem[i] will
     * have been signalled while goSem was reset, ensuring that
      the threads are pended or about to pend.
     * /
   DosEnterCritSec();
    synchRequested= TRUE;
    for (i = 0; i < 4; i++) {
       DosResetEventSem(threadWaitingSem[i], &eventCount);
    DosResetEventSem(goSem, &eventCount);
    DosExitCritSec();
```

```
* Wait for all movement to cease. When we come back,
         * we will know that all of the tasks passed the point
         ^{\ast}\, of posting to the threadWaitingSem while the goSem was
         * reset. Thus, we know that they are all waiting, or just about
         * to wait.
        DosWaitMuxWaitSem(
                        allThreadsWaitingSem, /* Semaphore handle
                                               /* Wait forever
                        SEM_INDEFINITE_WAIT,
                                                /* Unused
                        &lastEvent);
        synchRequested= FALSE;
        printf("All threads are stopped at stage %d\n", stage);
            Wait awhile
        DosSleep(2000);
       Print final message
    printf("We're done.\n");
}
        threadN()
        ThreadN prints messages. It uses handshaking with the main
    thread to stop and start.
void threadN(void *threadNum)
    long
                        me;
   ULONG
                       postCount;
   me= (long) threadNum;
    for (;;) {
       /*
* Post to my threadWaitingSem[]. This tells the main task
        * that I'm about to do my Wait.
        if (synchRequested) {
            * Note that we're in the synchronization code.
           DosResetEventSem(threadRunningSem[me], &postCount);
           /*

* Tell main thread we're about to pend
           DosPostEventSem(threadWaitingSem[me]);
    /**/
           DosSleep(10000);
                                   /* Open up window */
```

Figure 12.5 Continued

```
* Do the Wait. If the main task's reset goSem, then I'll pend.

* It also will have reset my threadWaitingSem, so my post will

* have told it that I'm here.

*/
DosWaitEventSem(goSem, SEM_INDEFINITE_WAIT);

/*

* Tell main thread we're running again.

*/
DosPostEventSem(threadRunningSem[me]);
}

printf("Thread %d crosses event at stage %d\n", me, stage);
}
```

There is one interesting note here. It might appear that we can remove synch Requested and the conditional based on it. Because we have threadRunning, we've effectively blocked out the race condition we had in prog12-3. This is not true. Without the conditional, we would have the following race condition:

We have a deadlock. We really need all of the machinery that we have. Or do we?

Not content to leave well enough alone, we have tweaked the main thread code a little bit. Notice the following: after we've muxwaited on threadRunningSem[] the auxiliary thread will not modify or examine threadWaitingSem[] or goSem. For this reason, we can eliminate the DosEnterCritSec()-DosExitCritSec() pair. We just need to set synchRequested to after the semaphores are reset:

```
muxwait on threadRunningSem[1-4]
for i from 1 to 4
  reset threadWaitingSem[i]
  endfor
reset goSem
synchRequested= FALSE;
muxwait on threadWaitingSem[1-4]
synchRequested= FALSE;
```

Program prog12-6.c, which is presented in Figure 12.6, is the finished product. It is robust and efficient, and it took only six tries.

Figure 12.6 prog12-6.c

```
prog12-6.c
        This program is our final solution to the problem
    of stopping and starting several threads.
        The program creates four threads that are to be stopped
    and started by the main thread. We would like for the main
    thread to know when they have stopped. In this version, we show
    that we can remove the DosEnterCritSec() and DosExitCritSec().
#include <stdio.h>
#include <stdlib.h>
#include <process.h>
#define INCL_DOSPROCESS
#define INCL_DOSSEMAPHORES
#include <os2.h>
#include "mt.h"
^{\prime*} ^{\star} Function declarations for the threads.
void threadN(void *);
   Declare our semaphores
*/
HEV
                        goSem;
                        threadWaitingSem[4];
HEV
HEV
                        threadRunningSem[4];
                        allThreadsWaitingSem;
HMUX
HMUX
                        allThreadsRunningSem;
char
                        synchRequested= FALSE;
   Declare a global that will indicate the stage we're at
long
                        stage;
        Main()
void main(int argc, char *argv[])
   TID
                        thread[4]:
   ULONG
                        i;
   ULONG
                        eventCount;
   ULONG
                        lastEvent:
   SEMRECORD
                        semaphoreList[4];
       Create all of the event semaphores
   DosCreateEventSem(
```

Figure 12.6 Continued

```
/* Local semaphore, no name */
/* Addr of semaphore handle */
                      &goSem,
                                            .
/* No options
                      FALSE):
                                            /* Initially not posted
for (i = 0; i < 4; i++) {
    DosCreateEventSem(
                      NULL.
                                            /* Local semaphore, no name */
                      &threadWaitingSem[i],/* Addr of semaphore handle */ 0, /* No options */
                                            /* Initially not posted
                      FALSE);
         Fill in the semaphore list for the MuxWait
    semaphoreList[i].hsemCur= (HSEM) threadWaitingSem[i];
    Create the MuxWait semaphore
DosCreateMuxWaitSem(
                                            /* Local semaphore, no name */
                      NULL,
                      &allThreadsWaitingSem,/* Addr of semaphore handle */
                      4, /* Three mutex semaphores */
semaphoreList, /* Array of semaphores */
DCMW_WAIT_ALL); /* Wait till all are free */
for (i = 0; i < 4; i++) {
    DosCreateEventSem(
                                             /* Local semaphore, no name */
                      &threadRunningSem[i],/* Addr of semaphore handle */ 0, /* No options */
                                            /* Initially posted
                      TRUE );
       Fill in the semaphore list for the MuxWait
     semaphoreList[i].hsemCur=\ (HSEM)\ threadRunningSem[i];
    Create the MuxWait semaphore
DosCreateMuxWaitSem(
                                            /* Local semaphore, no name */
                      &allThreadsRunningSem, /* Addr of semaphore handle */
                                   /* Three mutex semaphores */
                                            /* Array of semaphores
                       semaphoreList,
                                           /* Wait till all are free */
                      DCMW_WAIT_ALL);
    Start the other threads
for (i = 0; i < 4; i++) {
     thread[i]= _beginthread(
                                           /* Address of function
                      threadN.
                                           /* Don't really give stack
/* Size of stack needed
/* Message to thread
                       NULL,
                       STACK_SIZE,
                       (PVOID) i);
     }
```

for (stage= 1; stage <= 5; stage++) {

```
* Post to the semaphore allowing the tasks to run.
    DosPostEventSem(goSem);
        Let them run for a little bit
    DosSleep(400);
     * Wait for them to get out of the synchronization block
    DosWaitMuxWaitSem(
                    allThreadsRunningSem,
                    SEM_INDEFINITE_WAIT,
                    &lastEvent);
       And stop any movement.
       We have eliminated the Dos<Enter, Exit>CritSec(). To do
     * this, we have to reorder the statements a little. We set
     * synchRequested _last_. threadWaitingSem[] and goSem are
     * touched only by the auxiliary thread if synchRequested is
     * TRUE. Thus, we can reset them whithout fear while it's FALSE.
    for (i = 0; i < 4; i++) {
        DosResetEventSem(threadWaitingSem[i], &eventCount);
    DosResetEventSem(goSem, &eventCount);
    synchRequested= TRUE;
     * Wait for all movement to cease. When we come back,
     * we will know that all of the tasks passed the point
     ^{\star}\, of posting to the threadWaitingSem while the goSem was
    * reset. Thus, we know that they are all waiting, or just about
     * to wait.
     */
    DosWaitMuxWaitSem(
                    allThreadsWaitingSem,
                                            /* Semaphore handle
                    SEM_INDEFINITE_WAIT,
                                            /* Wait forever
                                            /* Unused
                    &lastEvent);
    synchRequested= FALSE;
    printf("All threads are stopped at stage %d\n", stage);
       Wait a while
   DosSleep(2000);
   Print final message
printf("We're done.\n");
```

Figure 12.6 Continued

```
threadN()
        ThreadN prints messages. It uses handshaking with the main
    thread to stop and start.
void threadN(void *threadNum)
    long
   ULONG
                        postCount;
    me= (long) threadNum;
    for (;;) {
           Post to my threadWaitingSem[]. This tells the main task
         * that I'm about to do my Wait.
        if (synchRequested) {
             ^{\star} Note that we're in the synchronization code.
            DosResetEventSem(threadRunningSem[me], &postCount);
             * Tell main thread we're about to pend
            DosPostEventSem(threadWaitingSem[me]);
    /**/
            DosSleep(10000);
                                      /* Open up window */
             ^{\star} Do the Wait. If the main task's reset goSem, then I'll pend.
             * It also will have reset my threadWaitingSem, so my post will
             * have told it that I'm here.
            DosWaitEventSem(goSem, SEM_INDEFINITE_WAIT);
                Tell main thread we're running again.
            DosPostEventSem(threadRunningSem[me]);
        printf("Thread %d crosses event at stage %d\n", me, stage);
}
```

Summary

Programming with threads is subtle and difficult. We have shown a simple problem which required a surprising amount of machinery to solve. When programming with threads, it is critical that you consider all possible scheduling sequences to assure that your program will work correctly all the time.

Chapter

13

The Producer/Consumer Problem

A standard problem in multithreading and multitasking literature is the so-called producer/consumer problem. The problem is this: thread A produces "widgets," thread B consumes widgets. We want B to get only valid widgets. We want B to block when there are no widgets, and we want it to unblock as soon as some widgets become available.

To give a concrete example (the example we will be using throughout this chapter), consider copying a file. The standard way to copy a file is to read a block from the input file, write it to the output file, and go back and do it again. It is easy to see that the elapsed time required to perform this file copy is the time required to read the input file plus the time required to write the output file. We can improve this performance by overlapping the reads and writes. Using multithreading to accomplish this, we find ourselves faced with the classical example of the producer/consumer problem. The reader "produces" buffers full of data by reading from the input file into memory. The writer "consumes" buffers full of data by writing them to the output file.

If the reader and writer were guaranteed to proceed at exactly the same rate, only two buffers would be required—one buffer from which the writer is writing, and one buffer into which the reader is reading. After each I/O, they would swap buffers. This is called, not surprisingly, *double-buffering*. In practice, reading and writing take different and varying amounts of time. Thus, it is helpful to have more than two buffers. Ideally, an infinite number of buffers could be used to ensure that the reader is never held up if the writer is slow. In practice, however, infinite memory is prohibitively expensive. Thus, one has to work with a finite number of buffers. As a result, the writer needs to return buffers to the reader for reuse. So, we really have two ongoing producer/consumer problems: the reader produces full buffers for consumption by the writer and the writer produces empty buffers for consumption by the reader.

Program prog13-1.c, which is presented in Figure 13.1, shows a first, rather brute force, approach to the problem. A circular array of buffers is maintained (although it's not really circular, we call it circular because, when a thread gets to the end of the array, it goes to the beginning again). Along with the array of buffers, an insertion point, a removal point, two semaphores for synchronization and a mutex semaphore are used. The insertion point indicates which is the next buffer to receive input from the reader thread. Data is written to buff[0], buff[1], etc., up to buff[NUM_BUFFS], then back to buff[0] again. The removal point indicates the next buffer to write data out of. Again, data is removed from buff[0], buff[1]...buff[NUM_BUFFS], buff[0], etc.

Figure 13.1 prog13-1.c

```
prog13-1.c
       This program does a file-to-file copy.
   This program is a classical example of using multithreading.
   It does a file-to-file copy using multibuffering to achieve
    overlapped I/O. The program has two threads: a reader and a
    writer. This allows reads and writes to be going on
    simulateously. An array of buffers is used to transmit blocks
    from the reader to the writer.
    This is our first implementation. We use fairly brute force
    techniques. We represent the communication channel between the
    threads using an array of buffers, an insertion point, a removal
   point, empty and full flags, and a mutex lock for the whole
   structure. This is a fairly classical solution to the problem.
    The structure of the two threads is very much complementary.
    The main thread consumes full buffers, writes them to a file
    and produces empty buffers. The reader thread consumes empty
    buffers, fills them from a file, and produces full buffers.
    This program is an example of the "producer/consumer" problem,
    a classical multithreading/multitasking problem.
#include <stdio.h>
#include cess.h>
#include <malloc.h>
#define INCL_DOSPROCESS
#define INCL_DOSSEMAPHORES
#define INCL_DOSFILEMGR
#include <os2.h>
#include "mt.h"
                                /* Number of buffers to use
#define NUM BUFFS
#define BUFF_SIZE
                        65535
                               /* Size of each buffer
   Function declarations for the thread.
void reader(void *):
```

Define the buffer and semaphores.

```
These definitions essentially constitute a communication channel
    between the reader and the writer.
 */
                       *buff[NUM BUFFS];
char
                                                 /* Array of buffers
ULONG
                        buffLen[NUM_BUFFS];
                                                 /* Buffer lengths
                                                 /* Insertion point
                        nextEmptyBuff;
int
                        nextFullBuff;
                                                 /* Removal point
int
                                                 /* Empty flag
                        ringBuffEmpty;
int
int
                        ringBuffFull;
                                                 /* Full flag
HMTX
                        ringBufferLock;
                                                 /* Mutex lock
HEV
                                                 /* Reader waits on this */
                        emptyBuffAvail;
                                                 /* Writer waits on this */
HEV
                        fullBuffAvail;
        Main()
void main(int argc, char *argv[])
    TID
                        readerTid;
    HFILE
                        inFile;
    HFILE
                        outFile:
    ULONG
                        i;
    ULONG
                        err:
    ULONG
                        bytesWritten;
    ULONG
                        actionTaken;
    ULONG
                        postCount;
     * First validate the command line
    * /
    if (argc != 3) {
        printf("Usage: %s <input pathname> <output pathname>\n", argv[0]);
        return;
        }
    err= DosOpen(
                        argv[1],
                        &inFile,
                        &actionTaken,
                        FILE NORMAL.
                        OPEN_ACTION OPEN IF EXISTS,
                        OPEN_FLAGS_SEQUENTIAL :
                            OPEN_SHARE_DENYNONE :
                            OPEN_ACCESS_READONLY,
                        NULL);
    if (err) {
        printf("Error: Unable to open input file \"%s\" (%d)\n",
                    argv[1],
                    err);
        return;
    err= DosOpen(
                        argv[2],
                        &outFile,
                        &actionTaken,
                        FILE_NORMAL,
                        OPEN_ACTION_CREATE_IF_NEW :
                            OPEN_ACTION_REPLACE_IF_EXISTS,
```

Figure 13.1 Continued

```
OPEN FLAGS SEQUENTIAL :
                        OPEN SHARE DENYNONE :
                        OPEN ACCESS WRITEONLY,
                    NULL):
if (err) {
    printf("Error: Unable to open output file \"%s\" (%d)\n",
                argv[2],
                err);
    return;
    }
   Now create the semaphores we need.
DosCreateMutexSem(
                    NULL.
                                        /* Local semaphore, no name */
                                        /* Addr of semaphore handle */
                    &ringBufferLock,
                                        /* No options
                                        /* Initially not owned
                                                                     */
                    FALSE):
DosCreateEventSem(
                    NULL.
                                        /* Local semaphore, no name */
                    &emptyBuffAvail,
                                        /* Addr of semaphore handle */
                                        /* No options
                    0.
                    FALSE):
                                        /* Initially not posted
DosCreateEventSem(
                                        /* Local semaphore, no name */
                                        /* Addr of semaphore handle */
                    &fullBuffAvail,
                                        /* No options
                    0.
                                        /* Initially not posted
                    FALSE):
   Allocate the buffers we need
for (i = 0; i < NUM_BUFFS; i++) {
    buff[i]= malloc(BUFF_SIZE);
   Initialize the ring buffer to indicate all empty buffers
 */
nextEmptyBuff= 0;
nextFullBuff= 0;
ringBuffEmpty= TRUE;
ringBuffFull= FALSE;
 * Start the reader thread:
readerTid= _beginthread(
                                        /* Address of function
                    NULL,
                                        /* Don't really give stack */
                                        /* Size of stack needed
                    STACK_SIZE,
                                        /* Message to thread
                    (PVOID) inFile);
    We now get down to the business of writing to the output file.
for (;;) {
        See if there are any buffers with contents. If not, wait
```

```
till there are.
   Note that we do not secure the lock before examining
 * ringBufferEmpty. There are two possibilities: We find it
    set and it is reset immediately thereafter. This is no problem
    because, when it is set, the fullBuffAvail semaphore will be
    posted, so we will not block. The other possibility is that
    we find it reset and it becomes set. This cannot happen because
    only this task can set the ringBufferEmpty flag.
if (ringBuffEmpty) {
    * Wait for notification of a full buffer
    DosWaitEventSem(fullBuffAvail, SEM INDEFINITE WAIT);
    DosResetEventSem(fullBuffAvail, &postCount);
    Here again, we use nextFullBuff without having the lock. This
    is OK, because only we modify it.
   A zero length buffer indicates EOF
if (buffLen[nextFullBuff] == 0) {
    break:
   Write the buffer out.
DosWrite(outFile, buff[nextFullBuff],
                buffLen[nextFullBuff], &bytesWritten);
   We now will update the status of the ring buffer.
   To do this we _must_ get the mutex lock.
DosRequestMutexSem(ringBufferLock, SEM INDEFINITE WAIT);
* Move to the next buffer
nextFullBuff++;
if (nextFullBuff >= NUM_BUFFS) {
    nextFullBuff= 0;
   See if the ring buffer is empty now
if (nextFullBuff == nextEmptyBuff) {
    ringBuffEmpty= TRUE;
^{\star} \, The ring buffer cannot be full because we've just removed a
   buffer, so, if it was formerly full, signal the other thread that
* there's room now.
if (ringBuffFull) {
```

Figure 13.1 Continued

```
ringBuffFull= FALSE;
            DosPostEventSem(emptyBuffAvail);
           OK, we're done. We can release the ring buffer lock now.
       DosReleaseMutexSem(ringBufferLock);
       We're done. Close the output file
   DosClose(outFile);
       Wait for reader to complete
   DosWaitThread(&readerTid, DCWW_WAIT);
        reader()
       Reader reads from the designated file, filling buffers until
   EOF, when it sends a zero length buffer. Its structure is very
   much complementary to the main thread which write the output
    file.
void reader(void *inFileV)
   HFILE
                        inFile;
   USHORT
                        done;
   ULONG
                        postCount;
    /*

* Start with a little type conversion
    inFile= (HFILE) inFileV;
        We now get down to the business of reading from the input file
    done= FALSE;
    while (!done) {
         ^{\star} See if there are any buffers with room. If not, wait
         * till there are.
         * Note that we do not require a lock before examining
         * ringBuffFull. The explanation is the same as that for
         * ringBuffEmpty in the main thread.
        if (ringBuffFull) {
            /*
```

```
* Wait for notification of a full buffer
            DosWaitEventSem(emptyBuffAvail, SEM_INDEFINITE_WAIT);
            DosResetEventSem(emptyBuffAvail, &postCount);
            }
            Read into the buffer
        DosRead(inFile, buff[nextEmptyBuff],
                        BUFF_SIZE, &buffLen[nextEmptyBuff]);
            If that was a zero length read, we've hit EOF (presumably)
            and we're done.
        if (buffLen[nextEmptyBuff] == 0) {
            done= TRUE;
            }
           Update the status of the ring buffer. To do this, we must get
         * the mutex lock.
        DosRequestMutexSem(ringBufferLock, SEM_INDEFINITE_WAIT);
           Move to the next buffer
         */
        nextEmptyBuff++;
        if (nextEmptyBuff >= NUM_BUFFS) {
            nextEmptyBuff= 0;
            See if the ring buffer is full now
        if (nextFullBuff == nextEmptyBuff) {
            ringBuffFull= TRUE;
            }
           The ring buffer cannot be empty because we've just added a
           buffer, so, if it was formerly empty, signal the other thread that
           there's data now.
        if (ringBuffEmpty) {
            ringBuffEmpty= FALSE;
            DosPostEventSem(fullBuffAvail);
           OK, we're done. We can release the ring buffer now.
        DosReleaseMutexSem(ringBufferLock);
       Close the file
    DosClose(inFile);
}
```

Note that we cannot always determine the state of the buffer from the insertion and removal points. If they are equal, it might be that all of the buffers are full or that all of the buffers are empty. For this reason, we keep two flags: ringBufferEmpty and ringBufferFull. These have the obvious meanings. When the ring buffer is full, each buffer in the buffer array contains data that has been read but not written. In this case, we cannot read any more data, because doing so would overwrite data that has not yet been written to the output file. When this happens, we want to block the reader until there are some empty buffers. Similarly, we want to block the writer when all of the buffers are empty. It does us no good to write empty blocks to the output file. For this purpose, we use event semaphores: one for blocking the writer when the buffers are empty, fullBuffAvail, and one for blocking the reader when the buffers are full, emptyBuffAvail.

The basic algorithm for the reader is this:

```
loop
if buffer is full
wait on emptyBuffAvail
read into next buffer
lock ring buffer
advance insertion point
if buffer was empty
signal fullBuffAvail
reset buffer empty flag
unlock ring buffer
endloop
```

It is necessary to get exclusive access to the ring buffer databases after the read. This is because we are updating several variables that need to be kept in a consistent state.

The writer has essentially the complementary task:

```
loop
  if buffer is empty
      wait on fullBuffAvail
  write from next buffer
  lock ring buffer
  advance removal point
  if buffer was full
      signal emptyBuffAvail
  reset buffer full flag
  unlock ring buffer
endloop
```

You can see how similar the two threads are. This is because each is consuming one resource and producing another.

In prog13-2.c, which is presented in Figure 13.2, we present a significantly more clever solution to the problem of copying a file. We make the observation that the writer needs to know only two things: what is the next buffer to write from and how many full buffers are available to write. The reader requires the complementary information: what is the next buffer to read into and how many empty buffers are available. Both threads know which buffer to process next based on how many buffers they've processed. Thus, the only information that needs to be communicated between the threads is the number of buffers available to each thread. Because event semaphores count the number of posts, we are able to use them to transmit this information.

Figure 13.2 prog13-2.c

```
prog13-2.c
        This program does a file-to-file copy.
    This program performs the same function as prog13-1. Here,
    however, we've gotten a great deal more clever. Event semaphores
    have post counts associated with them. We use these post counts
    to communicate between threads. The only data that is shared
    between the two threads is the array of buffers, and the two
    semaphores. Note that we have been able to dispense with
    mutex semaphores.
#include <stdio.h>
#include <malloc.h>
#include <process.h>
#define INCL_DOSPROCESS
#define INCL_DOSSEMAPHORES
#define INCL_DOSFILEMGR
#include <os2.h>
#include "mt.h"
                                /* Number of buffers to use
#define NUM BUFFS
                        65536 /* Size of each buffer
#define BUFF_SIZE
* Function declarations for the thread.
void reader(void *);
* Define the buffer and semaphores:
char
                       *buff[NUM_BUFFS];
ULONG
                       buffLen[NUM_BUFFS];
HEV
                        buffersEmpty;
HEV
                        buffersFull;
       Main()
void main(int argc, char *argv[])
   TID
                        readerTid:
   HFILE
                        inFile:
   HFILE
                        outFile:
   ULONG
                       numBuffersFull;
   ULONG
                       currBuffer;
   ULONG
                       i;
   ULONG
                       err;
   ULONG
                       bvtesWritten:
   ULONG
                       actionTaken;
```

Figure 13.2 Continued

```
First validate the command line
if (argc != 3) {
   printf("Usage: %s <input pathname> <output pathname>\n", argv[0]);
err= DosOpen(
                    argv[1],
                    &inFile,
                    &actionTaken,
                    FILE NORMAL,
                    OPEN_ACTION_OPEN_IF_EXISTS,
                    OPEN_FLAGS_SEQUENTIAL :
                        OPEN_SHARE_DENYNONE |
                        OPEN_ACCESS_READONLY,
                    NULL);
if (err) {
   printf("Error: Unable to open input file \"%s\" (%d)\n",
                argv[1],
                err);
    return;
    }
err= DosOpen(
                    argv[2],
                    &outFile,
                    &actionTaken,
                    FILE_NORMAL,
                    OPEN_ACTION_CREATE_IF_NEW :
                        OPEN_ACTION_REPLACE_IF_EXISTS,
                    OPEN_FLAGS_SEQUENTIAL :
                        OPEN SHARE DENYNONE !
                        OPEN_ACCESS_WRITEONLY,
                    NULL);
if (err) {
    printf("Error: Unable to open output file \"%s\" (%d)\n",
                argv[2],
                err);
    return;
    }
 ^{\star} Now create the semaphores we need.
DosCreateEventSem(
                                         /* Local semaphore, no name */
                    NULL,
                                         /* Addr of semaphore handle */
                    &buffersEmpty,
                                         /* No options
                    FALSE);
                                         /* Initially not posted
DosCreateEventSem(
                    NULL,
                                         /* Local semaphore, no name */
                                         /* Addr of semaphore handle */
                    &buffersFull,
                                         /* No options
                                         /* Initially not posted
                    FALSE):
   Initialize buffersEmpty so that it has 8 posts to indicate
```

```
* 8 empty buffers.
 * While we're at it, allocate the buffers
for (i = 0; i < NUM_BUFFS; i++) {
    buff[i]= malloc(BUFF_SIZE);
   DosPostEventSem(buffersEmpty);
* No initialization is required for buffersFull, because there are
 * no buffers full initially.
 * Start the reader thread:
readerTid= _beginthread(
                                        /* Address of function
                    reader,
                    NULL,
                                        /* Don't really give stack */
                    STACK SIZE.
                                       /* Size of stack needed
                    (PVOID) inFile);
                                       /* Message to thread
   We now get down to the business of writing to the output file.
numBuffersFull= 0;
currBuffer= 0;
for (;;) {
   /*
       See if there are any buffers with contents. If not, wait
    * till there are.
   if (numBuffersFull == 0) {
       DosWaitEventSem(buffersFull, SEM_INDEFINITE_WAIT);
        * The post count will give us the number of buffers
        * filled.
       DosResetEventSem(buffersFull, &numBuffersFull);
       }
       A zero length buffer indicates EOF
   if (buffLen[currBuffer] == 0) {
       break;
       }
    * Write the buffer out.
   DosWrite(outFile, buff[currBuffer],
                   buffLen[currBuffer], &bytesWritten);
    ^{\star} Advance to the next buffer and post another buffer empty.
   currBuffer++;
   if (currBuffer >= NUM_BUFFS) {
       currBuffer= 0:
   /*
```

Figure 13.2 Continued

```
* We've consumed a full buffer. Decrement the count of
         * full buffers.
         */
        numBuffersFull";
         * We've produced an empty buffer. Increment the count
         * of empty buffers.
        DosPostEventSem(buffersEmpty);
       Close the output file
    DosClose(outFile);
    * Wait for reader to complete
    DosWaitThread(&readerTid, DCWW_WAIT);
}
        reader()
        Reader reads from the designated file, filling buffers until
    EOF, when it sends a zero length buffer. Its structure is very
    much complementary to the main thread which write the output
    file.
void reader(void *inFileV)
                        inFile;
    HFILE
    ULONG
                        currBuffer;
    ULONG
                        numBuffersEmpty;
    USHORT
                        done;
    * Start with a little type conversion
    inFile= (HFILE) inFileV;
     ^{\ast} \, We now get down to the business of reading from the input file
    numBuffersEmpty= 0;
    currBuffer= 0;
    done= FALSE;
    while (!done) {
         ^{\ast}   
See if there are any buffers with room. If not, wait
         * till there are.
        if (numBuffersEmpty == 0) {
            DosWaitEventSem(buffersEmpty, SEM_INDEFINITE_WAIT);
```

```
The post count will give us the number of buffers
           available.
        DosResetEventSem(buffersEmpty, &numBuffersEmpty);
        Read into the buffer
    DosRead(inFile, buff[currBuffer],
                    BUFF_SIZE, &buffLen[currBuffer]);
        If that was a zero length buffer, we're done.
    if (buffLen[currBuffer] == 0) {
        done= TRUE;
        Advance to the next buffer and post another buffer filled.
    currBuffer++;
    if (currBuffer >= NUM_BUFFS) {
        currBuffer= 0;
        We've consumed an empty buffer, decrement the count.
    numBuffersEmpty";
        We've produced a full buffer. Increment that count.
    DosPostEventSem(buffersFull);
    Close the file
DosClose(inFile);
```

At any given point, the number of buffers that the writer has to process is its local variable numBuffersFull plus the post count of the event semaphore buffers Full. Every time the reader fills a buffer, it posts to buffersFull. When num BuffersFull goes to zero, the writer waits on buffersFull and resets it, obtaining the post count. It puts this post count into numBuffersFull. The writer decrements numBuffersFull after processing a buffer. Thus, the flow is this: the semaphore post count is incremented every time a buffer is filled.

Periodically, the post count is transferred to numBuffersFull. Every time a buffer is emptied, numBuffersFull is decremented. There are two reasons why we want to use a semaphore rather than simply using a memory location for buffersFull. First, incrementing (posting) a semaphore is an atomic action. Incrementing a memory location is not necessarily atomic and would require a mutex semaphore to insure atomicity. Second, and more important, a wait on a semaphore blocks when the

post count is zero. Note that the writer only waits on buffersFull if numBuffers Full is zero. Thus, it blocks exactly when there are no buffers available to process (numBuffersFull + postCount(buffersFull) = 0).

The reader functions in a precisely dual fashion. The number of empty buffers, buffers that have been read and written, or buffers that were never read into is num BuffersEmpty plus the post count of buffersEmpty. It is worth studying the code for some time to understand how all this works.

We show a third and still more clever solution in prog13-3.c, which is presented in Figure 13.3. Here, we've been able to dispense with all of our shared databases and semaphores. We replace them with two queues. One queue supplies empty buffers to the reader, and the other queue supplies full buffers to the writer. The reader takes buffers off of the empty buffer queue, reads into them, and writes them to the full buffer queue. The writer removes buffers from the full buffer queue, writes them to the output file, and returns them to the empty buffer queue.

The beauty of this solution is its simplicity. A quick look at the algorithm, and you know it's right. The other two solutions require a substantial amount of thought before you can believe they are correct. In point of fact, we really are begging the question in some sense. We are not getting around solving the producer/consumer problem, we merely are taking advantage of OS/2's queue facility, which is, in itself, a solution to the producer/consumer problem. However, it is instructive to see how much easier it is to use someone else's solution rather than growing your own.

The only drawback of using queues for intra-process communication is that OS/2 queues are global and named. Our example uses a fixed name. If you try to run two copies at the same time, you will run into problems, because the first one that you invoke will create the queues, and the second will fail because it can't create the queues—they already exist. This can be solved by creating unique queue names (for example, including the PID of the current process in the name), but it requires extra work.

Figure 13.3 prog13-3.c

```
This program does a file-to-file copy.

This program performs the same function as prog13-1 and prog13-2. Here, we've gotten more clever still. We've eliminated the array of buffers and the semaphores and replaced them with two queues. One is a queue of empty buffers, the other is a queue of full buffers. The reader pulls empty buffers off of the empty buffer queue and writes full buffers to the full buffer queue. The writer does the opposite.

No shared databases are required, nor are any semaphores required.
```

```
#include <stdio.h>
#include <malloc.h>
#include <process.h>
#define INCL_DOSPROCESS
#define INCL_DOSQUEUES
#define INCL_DOSFILEMGR
#include <os2.h>
#include "mt.h"
#define NUM_BUFFS
                              /* Number of buffers to use
                       65536 /* Size of each buffer
#define BUFF_SIZE
 * Function declarations for the thread.
void reader(void *);
 * Define the queues
HQUEUE
                        emptyBuffers;
HQUEUE
                        fullBuffers;
        Main()
void main(int argc, char *argv[])
{
                        readerTid:
    TID
    HFILE
                        inFile;
    HFILE
                       outFile;
    ULONG
                       i;
    ULONG
                       err;
                        bytesWritten;
    ULONG
    ULONG
                       actionTaken;
    REQUESTDATA
                       requestID;
    BYTE
                        priority;
    PVOID
                        buffAddr;
    ULONG
                        buffLen;
       First validate the command line
    if (argc != 3) {
        printf("Usage: %s <input pathname> <output pathname>\n", argv[0]);
        return;
    err= DosOpen(
                        argv[1],
                        &inFile,
                        &actionTaken,
                        0,
                        FILE_NORMAL,
                        OPEN_ACTION_OPEN_IF_EXISTS,
                        OPEN_FLAGS_SEQUENTIAL :
```

Figure 13.3 Continued

```
OPEN_SHARE_DENYNONE |
                        OPEN_ACCESS_READONLY,
                    NULL);
if (err) {
    printf("Error: Unable to open input file \"%s\" (%d)\n",
                argv[1],
                err):
    return;
    }
err= DosOpen(
                    argv[2],
                    &outFile,
                    &actionTaken.
                    FILE NORMAL,
                    OPEN_ACTION_CREATE_IF_NEW :
                        OPEN_ACTION_REPLACE_IF_EXISTS,
                    OPEN_FLAGS_SEQUENTIAL :
                        OPEN_SHARE_DENYNONE |
                        OPEN_ACCESS_WRITEONLY,
                    NULL);
if (err) {
    printf("Error: Unable to open output file \"%s\" (%d)\n",
                argv[2],
                err);
    return;
    }
   Now create the queues we need
DosCreateQueue(
                                        /* Queue handle
                    &emptyBuffers,
                    QUE_FIFO,
                                        /* First In First Out
                    "\\QUEUES\\prog13-3\\emptyBuffers");
                                       /* Queue name (mandatory)
DosCreateQueue(
                    &fullBuffers.
                                        /* Queue handle
                    QUE FIFO.
                                        /* First In First Out
                    "\\QUEUES\\prog13-3\\fullBuffers");
                                        /* Queue name (mandatory)
   Initialize emptyBuffers so that it has NUM_BUFFS empty buffers
   queued up.
*/
for (i = 0; i < NUM_BUFFS; i++) {
    buffAddr= malloc(BUFF_SIZE);
    DosWriteQueue(
                    emptyBuffers,
                                        /* emptyBuffers Queue
                    0,
                                        /* No request ID
                    BUFF_SIZE,
                                       /* Length of data
                    buffAddr,
                                       /* Buffer address
                    0):
                                        /* Priority (none)
    }
   No initialization is required for buffersFull, because there are
```

```
no buffers full initially.
    Start the reader thread:
readerTid= _beginthread(
                                             /* Address of function
/* Don't really give stack
/* Size of stack needed
/* Message to thread
                        reader,
                       NULL,
                       STACK_SIZE,
                        (PVOID) inFile);
    We now get down to the business of writing to the output file.
for (;;) {
    /*
* Sit and wait for a full buffer to come our way:
    DosReadQueue(
                        fullBuffers.
                                               /* Queue handle
                                               /* User Request ID
                       &requestID,
                                              /* Length of message
/* Address of message
                       &buffLen,
                       &buffAddr,
                                             /* Get first element
/* Wait for a request
/* Message priority
/* Sem handle (unused)
                       DCWW_WAIT,
                       &priority,
                       NULL);
         A zero length buffer indicates EOF
     if (buffLen == 0) {
         break;
         }
     /*

* Write the buffer out.
     DosWrite(outFile, buffAddr, buffLen, &bytesWritten);
     ^{\prime*} ^{\star} Now send the buffer back. It's empty.
    DosWriteQueue(
                                             /* emptyBuffers Queue
                        emptyBuffers,
                                              /* No request ID
                        BUFF_SIZE,
                                             /* Length of data
                                              /* Buffer address
                       buffAddr,
                                              /* Priority (none)
                       0):
   Close the output file
DosClose(outFile);
* Wait for reader to complete
DosWaitThread(&readerTid, DCWW_WAIT);
```

}

Figure 13.3 Continued

```
reader()
        Reader reads from the designated file, filling buffers until
   EOF, when it sends a zero length buffer. Its structure is very
   much complementary to the main thread which write the output
    file.
void reader(void *inFileV)
                        inFile:
    HFILE
   USHORT
                        done;
    REQUESTDATA
                        requestID;
    BYTE
                        priority;
    PVOID
                        buffAddr:
    ULONG
                        buffLen;
       Start with a little type conversion
    inFile= (HFILE) inFileV;
       We now get down to the business of reading from the input file
    done= FALSE;
    while (!done) {
        /*
* Wait till we get an empty buffer:
        DosReadQueue(
                        emptyBuffers,
                                           /* Queue handle
                        &requestID,
                                            /* User Request ID
                                           /* Length of message
                        &buffLen,
                                           /* Address of message
                        &buffAddr,
                                            /* Get first element
                        DCWW_WAIT,
                                           /* Wait for a request
                                           /* Message priority
/* Sem handle (unused)
                        &priority,
                        NULL):
            Read into the buffer
        DosRead(inFile, buffAddr, BUFF_SIZE, &buffLen);
         * If that was a zero length buffer, we're done.
        if (buffLen == 0) {
            done= TRUE;
            }
         * It's a full buffer now. Put it on the full buffer queue.
        DosWriteQueue(
```

```
fullBuffers, /* emptyBuffers Queue */
0, /* No request ID */
buffLen, /* Length of data */
buffAddr, /* Buffer address */
0); /* Priority (none) */

/*

* Close the file
*/
DosClose(inFile);
```

Summary

The producer/consumer problem is a classical problem of multithreading. We have presented three solutions. The simplest solution uses queues. This is not surprising since queues are explicitly designed to solve the producer/consumer problem.

0.0



Chapter

14

A Complex Example

In this chapter, we present a reasonably complex example of using many multithreading facilities in one program. The primary goal of the program is to copy a series of files from the current directory to another directory. The secondary goal is to use as much multithreading as possible to solve the problem.

To this end, program prog14-1.c, which is presented in Figure 14.1, creates a large number of "server" threads. These threads are all equivalent and take requests from a single request queue. There are three requests that are sent to server threads: "read a file and write it to a buffer channel," "read from a buffer channel and write to a file," and "terminate."

Figure 14.1 prog14-1.c

prog14-1.c

This program copies a series of files from the current directory to another directory.

This program illustrates heavy multithreading. We copy all files matching a template from the current directory to a specified directory. To overlap this process as much as possible, we use server threads. The server threads take three requests: read a file, write a file, and terminate. The main thread finds all of the files matching the specified template. For each one, it creates a "buffer channel" and two requests. One request to read the file and write the contents into the buffer channel and another to read from the buffer channel and write to a new file. A buffer channel is much like the ring buffer scheme used in prog13-2. The major difference is that the buffer pool is global and is implemented using a queue.

Figure 14.1 Continued

```
#include <stdio.h>
#include <string.h>
#include <malloc.h>
#include <process.h>
#define INCL DOSPROCESS
#define INCL_DOSSEMAPHORES
#define INCL DOSQUEUES
#define INCL_DOSFILEMGR
#define INCL_DOSERRORS
#include <os2.h>
#include "mt.h"
#define NUM_BUFFS 32 /* Total number of buffers to use #define MAX_BUFFS_PER_FILE 8 /* Max buffer for each file #define BUFF_SIZE 65536 /* Size of each buffer #define NUM_SERVERS 16 /* Number of server tasks
 * Define the request types
#define REQ_FILE_READ 1 /* Read a file and write to buff chan*/
#define REQ_FILE_WRITE 2 /* Read from buff chan and write file*/
#define REQ_TERMINATE 3 /* Go bye bye */
 * Define a thread-to-thread buffer channel
typedef struct {
                            *buff[MAX_BUFFS_PER_FILE];
     char
     ULONG
                           buffLen[MAX_BUFFS_PER_FILE];
     HFV
                            fullBuffers:
     HEV
                             emptyBuffers;
     } BUFF CHAN:
     Define a request
typedef struct {
                          *fileName;
     char
     BUFF_CHAN
                         *buffChan;
     } FILE REQUEST;
 * Function declarations
void serverThread(void *);
void readFile(char *fileName, BUFF_CHAN *buffChan);
void writeFile(char *fileName, BUFF_CHAN *buffChan);
 * Define the two global queues:
*/
```

```
203
```

```
HOUFUE
                        freeBuffers:
                                            /* Free memory buffers
HOUEUE
                        requests:
                                            /* Request queue
       Main()
void main(int argc, char *argv[])
    ULONG
                        i:
   ULONG
                        err;
   ULONG
                        tid[NUM_SERVERS];
    BUFF CHAN
                       *newBuffChan:
    FILE_REQUEST
                       *newFileRequest;
    char
                       *newBuffer:
    HDTR
                        dirHandle;
    FILEFINDBUF3
                        findPk:
   ULONG
                        findCt;
       First validate the command line
    if (argc != 3) {
       printf("Usage: %s <input template> <output directory>\n", argv[0]);
        return;
        }
       Before we do anything else, make sure we have enough file
     * handles. We can open as many files as we have server threads.
     * Add 10 handles to be on the safe side.
    DosSetMaxFH(NUM_SERVERS+10);
       Create the request and freeBuffer queues
   DosCreateQueue(
                                            /* Queue handle
                        &freeBuffers,
                                            /* First In First Out
                        QUE_FIFO,
                        "\\QUEUES\\prog14-1\\freeBuffers");
                                            /* Queue name (mandatory)
   DosCreateQueue(
                        &requests,
                                            /* Queue handle
                        QUE FIFO.
                                            /* First In First Out
                                                                         */
                        "\\QUEUES\\prog14-1\\requests");
                                            /* Queue name (mandatory)
       Now create the server threads. All of the server threads are
      equivalent. Note that they all have the same starting address.
    for (i = 0; i < NUM\_SERVERS; i++) {
       tid[i]= _beginthread(
                                            /* Address of function
                        serverThread,
                        NULL.
                                            /* Don't really give stack
                                            /* Size of stack needed
                        STACK SIZE,
```

Figure 14.1 Continued

```
(PVOID) 0);
                                     /* Message to thread
    }
   Create and make available NUM BUFFS buffers:
for (i = 0; i < NUM_BUFFS; i++) {
    newBuffer= malloc(BUFF_SIZE);
    DosWriteQueue(
                    freeBuffers,
                                      /* Request Queue
                                      /* No request code
                                      /* Length of data
                    BUFF SIZE,
                                      /* Buffer address
                    newBuffer,
                                       /* Priority (none)
    }
    We're set to start. Let's get the file directory going
dirHandle= HDIR_CREATE;
findCt= 1;
err= DosFindFirst(
                    argv[1],
                    &dirHandle.
                    FILE ARCHIVED,
                    (PVOID) &findPk.
                    sizeof(findPk),
                   &findCt,
                   FIL_STANDARD);
   Now loop over the files that match the template
while (err == NO_ERROR) {
    * Allocate and initialize a BUFF_CHAN:
    newBuffChan= malloc(sizeof(BUFF_CHAN));
       Now create the semaphores we need.
    DosCreateEventSem(
                                        /* Local semaphore, no name */
                    &newBuffChan->emptyBuffers,
                                       /* Addr of semaphore handle */
                                        /* No options
                                       /* Initially not posted
                    FALSE):
    DosCreateEventSem(
                                       /* Local semaphore, no name */
                    &newBuffChan->fullBuffers,
                                      /* Addr of semaphore handle */
                                      /* No options
                    FALSE):
                                      /* Initially not posted
       Initialize emptyBuffers so that it has MAX_BUFFS_PER_FILE
       posts to indicate MAX_BUFFS_PER_FILE empty buffers.
```

```
for (i= 0; i < MAX_BUFFS_PER_FILE; i++) {
    DosPostEventSem(newBuffChan->emptyBuffers);
   No initialization is required for fullBuffers, because there are
   no buffers full initially.
   Now construct the read request
*/
newFileRequest= malloc(sizeof(FILE_REQUEST));
newFileRequest->fileName= malloc(strlen(findPk.achName)+1);
strcpy(newFileRequest->fileName, findPk.achName);
newFileRequest->buffChan= newBuffChan;
* Enqueue the request
DosWriteQueue(
                                     /* Request Queue
                requests,
                                    /* Termination
                REQ_FILE_READ,
                sizeof(FILE_REQUEST),/* Length of data
                newFileRequest,
                                  /* Buffer address
                                     /* Priority (none)
                0);
   Construct the write request
newFileRequest= malloc(sizeof(FILE_REQUEST));
newFileRequest->fileName= malloc(strlen(findPk.achName) +
                                    strlen(argv[2]) + 2);
strcpy(newFileRequest->fileName, argv[2]);
strcat(newFileRequest->fileName, "\\");
strcat(newFileRequest->fileName, findPk.achName);
newFileRequest->buffChan= newBuffChan;
   Enqueue the request
DosWriteQueue(
                requests,
                                     /* Request Queue
                                    /* Termination
                REQ_FILE_WRITE,
                sizeof(FILE_REQUEST),/* Length of data
                                    /* Buffer address
                newFileRequest,
                                     /* Priority (none)
                0);
   We've got the file started. Let's go on to the next
   file.
findCt= 1;
err= DosFindNext(
                dirHandle,
                (PVOID) &findPk,
                sizeof(findPk),
                &findCt);
```

Figure 14.1 Continued

```
* Send enough termination requests to stop all of the threads.
    for (i = 0; i < NUM_SERVERS; i++) {
        DosWriteQueue(
                                            /* Request Queue
                        requests.
                                            /* Termination
/* Length of data
/* Buffer address
                        REQ TERMINATE,
                        NULL,
                                            /* Priority (none)
                        0);
        }
       Now wait for each of them to finish
    for (i = 0; i < NUM_SERVERS; i++) {
        DosWaitThread(&tid[i], DCWW WAIT);
}
        serverThread()
        All of the server threads take requests from a single
    request queue. Because they are all equivalent, it is irrelevant
    which thread gets a given request. The three requests are to
    read a file into a buffer channel, write a file from a buffer
    channel, and to terminate.
void serverThread(void *p)
{
    REQUESTDATA
                        requestID:
    UCHAR
                        priority;
    ULONG
                        requestLen;
    FILE_REQUEST
                       *request;
        Just loop, waiting for and processing requests.
    for (;;) {
         * Pull a request off of the request queue
        DosReadQueue(
                                             /* Queue handle
                         requests,
                                             /* User Request ID
                         &requestID,
                                            /* Length of message
                         &requestLen,
                         (PVOID *) &request, /* Address of message
                                           /* Get first element
                                            /* Wait for a request
                         DCWW WAIT,
                                            /* Message priority
                         &priority,
                                            /* Sem handle (unused)
                         NULL);
        /*
* Now service it appropriately
```

```
switch (requestID.ulData) {
        case REQ_FILE_READ:
            /*
* Call readFile to read the file into the buffer channel
            readFile(request->fileName, request->buffChan);
             * Now just free the request memory.
            free(request);
            break;
        case REQ_FILE_WRITE:
            ^{\prime*} ^{\star} Call writeFile to read out of the buffer channel into the
             * file.
            writeFile(request->fileName, request->buffChan);
             * We're done now. Free the semaphores and the buffer channel.
            DosCloseEventSem(request->buffChan->fullBuffers);
            DosCloseEventSem(request->buffChan->emptyBuffers);
            free(request->buffChan);
             * Now just free the request memory
            free(request);
            break;
        case REQ TERMINATE:
             * Just return. This terminates the thread.
            return;
}
        readFile()
        This procedure reads a file and writes it to a buffer
    channel. It's algorithm is essentially that of prog13-2.
    The major difference is that empty buffers are secured from the
    freeBuffers queue.
void readFile(fileName, buffChan)
                       *fileName:
    char
    BUFF CHAN
                       *buffChan:
    HFILE
                        inFile;
    ULONG
                        currBuffer:
    ULONG
                        numBuffersEmpty;
```

Figure 14.1 Continued

```
USHORT
                    done:
ULONG
                    err:
                    actionTaken;
ULONG
REQUESTDATA
                    requestID;
UCHAR
                    priority;
ULONG
                    buffLen;
char
                   *buffAddr:
* Start by opening the input file
actionTaken= 0;
inFile= 0:
err= DosOpen(
                    fileName.
                    &inFile.
                    &actionTaken,
                    FILE_NORMAL,
                    OPEN_ACTION_OPEN_IF_EXISTS,
                    OPEN_FLAGS_SEQUENTIAL :
                        OPEN_SHARE_DENYNONE |
                        OPEN_ACCESS_READONLY,
                    NULL);
if (err) {
    printf("Error: Unable to open input file \"%s\" (%d)\n",
                fileName,
                err);
       Wake up the waiter and indicate EOF.
    buffChan->buff[0]= NULL;
    buffChan->buffLen[0]= 0;
    DosPostEventSem(buffChan->fullBuffers);
    return:
^{\prime} ^{\star} We now get down to the business of reading from the input file
numBuffersEmpty= 0;
currBuffer= 0;
done= FALSE;
while (!done) {
     * See if there are any spots in the ring buffer with room. If
       not, wait till there are.
    if (numBuffersEmpty == 0) {
        DosWaitEventSem(buffChan->emptyBuffers, SEM_INDEFINITE_WAIT);
         * The post count will give us the number of buffers
         * available.
        DosResetEventSem(buffChan->emptyBuffers, &numBuffersEmpty);
```

```
Now wait for a free buffer
        DosReadQueue(
                                            /* Queue handle
                        freeBuffers,
                                            /* User Request ID
                        &requestID,
                                            /* Length of message
                        &buffLen,
                        (PVOID *) &buffAddr,/* Address of message
                                            /* Get first element
                                            /* Wait for a request
                        DCWW WAIT,
                                            /* Message priority
                        &priority,
                                            /* Sem handle (unused)
                        NULL);
        /*
* Read into the buffer
        err= DosRead(inFile, buffAddr, BUFF_SIZE,
                        &buffChan->buffLen[currBuffer]);
            Put the buffer address in the channel.
        buffChan->buff[currBuffer]= buffAddr;
         * If that was a zero length buffer, we're done.
        if (buffChan->buffLen[currBuffer] == 0) {
            done= TRUE;
           Advance to the next spot in the channel and post
            another buffer filled.
        currBuffer++;
        if (currBuffer >= MAX_BUFFS_PER_FILE) {
            currBuffer= 0;
            }
       numBuffersEmpty";
       DosPostEventSem(buffChan->fullBuffers);
       Close the file
   DosClose(inFile);
}
       writeFile()
       This procedure writes a file taking the data from a buffer
   channel. It's algorithm is essentially that of prog13-2.
   The major difference is that empty buffers are returned to the
   freeBuffers queue.
```

/*

```
*fileName:
    char
    BUFF_CHAN
                       *buffChan:
{
   HFILE
                        outFile;
   ULONG
                        numBuffersFull;
   ULONG
                        currBuffer:
   ULONG
                        err:
   ULONG
                        bytesWritten;
   ULONG
                        actionTaken:
    int
                        done:
    * Open the output file
    actionTaken= 0;
    outFile= 0;
    err= DosOpen(
                        fileName.
                        &outFile.
                        &actionTaken.
                        FILE NORMAL.
                        OPEN_ACTION_CREATE_IF_NEW :
                             OPEN_ACTION_REPLACE_IF_EXISTS,
                        OPEN_FLAGS_SEQUENTIAL !
                             OPEN_SHARE_DENYNONE |
                             OPEN_ACCESS_WRITEONLY,
                        NULL);
        printf("Error: Unable to open output file \"%s\" (%d)\n",
                    fileName.
                    err);
        return;
        }
     ^{\star} We now get down to the business of writing to the output file.
    numBuffersFull= 0;
    currBuffer= 0;
    done= FALSE;
    while (!done) {
         ^{\star}   
See if there are any buffers with contents. If not, wait
         * till there are.
        if (numBuffersFull == 0) {
            DosWaitEventSem(buffChan->fullBuffers, SEM_INDEFINITE_WAIT);
             * The post count will give us the number of buffers
             * filled.
            DosResetEventSem(buffChan->fullBuffers, &numBuffersFull);
```

```
A zero length buffer indicates EOF
    if (buffChan->buffLen[currBuffer] == 0) {
        done= TRUE;
    else {
           Write the buffer out.
        DosWrite(outFile, buffChan->buff[currBuffer],
                    buffChan->buffLen[currBuffer], &bytesWritten);
        }
        See if we were passed a buffer. If not don't try to
       re-queue it.
    if (buffChan->buff[currBuffer]) {
           Return the buffer to the buffer pool
        DosWriteQueue(
                    freeBuffers.
                                        /* Request Queue
                                        /* No request code
                                        /* Length of data
                    BUFF SIZE.
                    buffChan->buff[currBuffer],
                                       /* Buffer address
                                        /* Priority (none)
        }
        Advance to the next buffer and post another buffer empty.
    currBuffer++;
    if (currBuffer >= MAX_BUFFS_PER_FILE) {
        currBuffer= 0;
    numBuffersFull";
    DosPostEventSem(buffChan->emptyBuffers);
   Close the output file
DosClose(outFile):
```

A buffer channel is a structure that encapsulates the ring buffer concept used in prog12-2.c. It contains an array of buffer pointers and two semaphores. It functions as in prog12-2 with one minor exception: empty buffers are pulled from a global queue. This allows all of the server threads to share a common pool of buffers.

Without this feature, buffers might be wasted by threads that are copying small files and do not require many buffers. The main thread gets the files matching the template argument. For each file, it creates a buffer channel, then queues two requests: one to read the input file and write it to the buffer channel and another to read from the buffer channel and write to the output file. A pair of server threads is used for each file copy operation.

212 Real Multithreading Programs

One important concept illustrated here is that of reentrancy. A procedure is reentrant if it can function properly when being executed by more than one thread simultaneously. A procedure that uses global data or static local data is suspect. Synchronization in the form of mutex semaphores probably is required. Note that the server threads use only local variables. Because local variables are maintained on the stack, each thread has its own copies. Even though there are many threads executing the same code, the data that they are operating on is distinct. Thus, server-Thread is a reentrant procedure.

It also is important to note that all of the server threads read from the same request queue. While processes other than the creator of the queue cannot read from a queue, any thread in the creator process can read from the queue. What's more, it works as one would expect: each request is sent to exactly one thread. It will not happen that two threads wake up with the same request. This feature of queues also is used in the program to implement a shared buffer pool. One queue contains all of the free buffers, and any thread that needs one reads from the queue to secure a buffer.

Summary

We have presented a heavily multithreaded program. It uses a variety of multithreading tools including queues and event semaphores. Studying this program will help you to appreciate how the various elements of multithreading can be combined into useful applications.

Chapter

15

Event-Driven Input

In this chapter, we illustrate how queues can be used to build a program that uses event-driven input. In Presentation Manager programs, all input comes as events, or messages to window procedures. In character-mode applications, however, there are two separate sets of calls for obtaining input from the keyboard and input from the mouse. In this program, we present a method of merging these two input streams.

A typical approach used in DOS programs for merging keyboard and mouse input is polling. The program sits in a loop peeking to see whether there's keyboard input, then peeking to see whether there's mouse input. If it discovers input, it removes the input and generates an event. The problem with this approach in OS/2 is that the polling wastes CPU time. Instead, we want our program to be blocked when there is no input and unblocked when an input event occurs.

Our solution is simple. In program prog15-1.c, which is presented in Figure 15.1, we use three threads and one event queue. One thread is assigned to reading mouse events. It uses a blocking mouse read call. When a mouse event occurs, it awakens and creates a mouse event structure. It then writes this event to the event queue. Another thread reads from the keyboard. When a key is struck, it wakes up and creates a keystroke event structure. It writes this structure to the event queue. The main thread reads from the event queue and processes the events in whatever way it sees fit.

Here, we are doing essentially the same thing that we've done in chapters 12 and 13. We're using multithreading to achieve overlapped I/O. In this case, we can read both the keyboard and the mouse at the same time, whereas, in chapters 12 and 13, we were reading one file and writing another at the same time.

In this application, we are tracking the mouse using a character as the cursor. Striking a key assigns that key to be the cursor. Dragging button 1 paints the cursor character along the mouse's path. Dragging button 2 erases the screen cells in the mouse's path. Alt—X terminates the program. The attribute being used to write to the screen can be changed using Alt—D and Alt—U. Alt—D decrements the attribute value, and Alt—U increments it.

Figure 15.1 prog15-1.c

```
prog15-1.c
    This program demonstrates using queues to get "event driven"
input processing.
    The main thread creates two secondary threads. One reads
the mouse queue and one reads the keyboard. A queue is created.
Both the mouse thread and the keyboard thread encode their
events into structures and add the events to the single queue.
The main thread reads this queue and is able to get a merged
stream of input events.
    This particular program tracks the mouse, using a character
as the mouse cursor. When a key is struck that character
becomes the new mouse cursor. When the mouse is dragged with
button 1 depressed, the current cursor character is painted in
the mouse's path. When the mouse is dragged with button 2
depressed, the characters in the mouse's path are erased.
There are three function keys: Alt-X terminates the program.
Alt-U increments the attribute of the mouse cursor and painting.
Alt-D decrements the attribute.
```

```
#include <stdio.h>
#include <stdlib.h>
#include <process.h>
#define INCL DOSPROCESS
#define INCL_DOSQUEUES
#define INCL_KBD
#define INCL_MOU
#define INCL_VIO
#include <os2.h>
#include "mt.h"
 * Define the input event types
#define
            EVENT_KEYBOARD
                                1
            EVENT_MOUSE
#define
                                2
#define
            EVENT_INIT
   Define a few scan codes for fun
#define X_KEY
                    45
#define U KEY
                    22
#define D_KEY
                    32
 * Define the event descriptors
typedef struct {
    UCHAR
                        ascii:
    UCHAR
                        scan;
```

```
USHORT
                       shiftKevs:
    } KEYSTROKE;
typedef struct {
    USHORT
                       row;
    USHORT
                      col;
    USHORT
                       state;
    } MOUSEEVENT;
 * Function declarations for the threads.
void keyboardThread(void *);
void mouseThread(void *);
/*
* processEvent() is the routine that handles input events. It is
 * akin to a window procedure in a PM program.
int processEvent(ULONG, PVOID);
/*
* Declare the input queue handle
HQUEUE
                       inputQ;
       Main()
void main(int argc, char *argv[])
{
    TID
                       mouseTid;
   TID
                       keyboardTid;
    REQUESTDATA
                       requestID;
    BYTE
                       priority;
    PVOID
                       buffAddr;
   ULONG
                       buffLen;
    int
                       done;
    * Create the input event queue
   DosCreateQueue(
                                           /* Queue handle
                       &inputQ,
                       QUE_FIFO,
                                           /* First In First Out
                       "\\QUEUES\\prog15-1\\inputQ");
                                          /* Queue name (mandatory)
    /*

* Start the keyboard thread.
    keyboardTid= _beginthread(
                                        /* Address of function
                       keyboardThread,
                                                                      */
                       NULL,
                                         /* Don't really give stack */
                                          /* Size of stack needed
                       STACK_SIZE,
```

Figure 15.1 Continued

```
(PVOID) 0):
                                            /* Message to thread
        Start the mouse thread.
   mouseTid= _beginthread(
                                            /* Address of function
                         mouseThread,
                                             /* Don't really give stack
                         NULL.
                                             /* Size of stack needed
                         STACK SIZE,
                                              /* Message to thread
                         (PVOID) 0);
    ^{\prime*} ^{\ast} Send an init event to allow the event procedure to initialize.
    processEvent(EVENT_INIT, NULL);
     * Loop reading events from the queue and processing them.
    done= FALSE;
    while (!done) {
        /*

* Read the next event from the queue.
        DosReadQueue(
                                              /* Queue handle
                         inputQ,
                                            /* User Request ID
                         &requestID,
                         &buffLen,
                                            /* Length of message
                         &buffAddr,
                                            /* Address of message
                                             /* Get first element
/* Wait for a request
/* Message priority
/* Sem handle (unused)
                         DCWW WAIT.
                         &priority,
                         NULL);
            Process the event. processEvent() returns TRUE when it
            decides the time has come to terminate.
        done= processEvent(requestID.ulData, buffAddr);
        free(buffAddr);
}
        processEvent()
        This procedure implements the mouse tracking etc. It takes
    mouse and keyboard events. Various applications could be
    implemented by changing this procedure. The rest of the program
    is fairly generic.
int processEvent(eventType, event)
    ULONG
                         eventType;
    PVOID
                         event;
{
```

```
static USHORT
                      row, col;
static UCHAR
                      backupCell[2]:
static UCHAR
                      key;
static UCHAR
                      attrib;
UCHAR
                     currCell[2];
KEYSTROKE
                     *keyboardEvent;
MOUSEEVENT
                     *mouseEvent;
USHORT
                     length;
 * See what kind of event we've got.
switch (eventType) {
case EVENT_INIT:
   /*

* Get things set up. Pretend we're at row 0, column 0. We need

* to have a current position and its content for when we get
     * another event.
    row=0;
    col= 0;
    * Get the contents at 0,0
    length= 2;
    VioReadCellStr(backupCell, &length, row, col, 0);
    ^{\prime*} \, Start the mouse cursor as "X" in the standard attribute
    key= "X";
    attrib= 0x07;
    break:
case EVENT_MOUSE:
    mouseEvent= (MOUSEEVENT *) event;
    ^{\prime} ^{\star} OK, see what kind of mouse event it is
    if (mouseEvent->state & MOUSE_MOTION) {
        /*
 * The mouse just moved. Restore it's former location...
        VioWrtCellStr(backupCell, 2, row, col, 0);
        /*
* ... backup its new location ...
        row= mouseEvent->row;
        col= mouseEvent->col;
        length= 2;
        VioReadCellStr(backupCell, &length, row, col, 0);
            ... and write the cursor to the new location
        currCell[0]= key;
        currCell[1]= attrib;
```

Figure 15.1 Continued

```
VioWrtCellStr(currCell, 2, row, col, 0);
   else if (mouseEvent->state & MOUSE_BN1_DOWN !!
               mouseEvent->state & MOUSE_MOTION_WITH_BN1_DOWN) {
        * We're dragging the mouse with button 1 down.
        * Restore the previous location...
       VioWrtCellStr(backupCell, 2, row, col, 0);
           ...and update the new location
       row= mouseEvent->row:
       col= mouseEvent->col:
        * Rather than reading the new location, we set the backup
        * to what we want to paint. This way, when we move, the paint
        * cell will be written to the current location.
       backupCell[0]= key;
       backupCell[1]= attrib;
        * And write the cursor character
       currCell[0]= key;
       currCell[1]= attrib;
       VioWrtCellStr(currCell, 2, row, col, 0);
   else if (mouseEvent->state & MOUSE BN2 DOWN !!
               mouseEvent->state & MOUSE_MOTION_WITH_BN2_DOWN) {
        * We're dragging button 2. Same thing as button 1 except
        * that, rather than painting with the current key, we paint
        * with spaces.
       VioWrtCellStr(backupCell, 2, row, col, 0);
       row= mouseEvent->row;
       col= mouseEvent->col;
        * We set the backup to " ", so when we move, we'll restore
        * the space rather than what was there.
       backupCell[0]= " ";
       backupCell[1]= attrib:
       currCell[0]= key;
       currCell[1]= (attrib >> 4) ! (attrib << 4);</pre>
       VioWrtCellStr(currCell, 2, row, col, 0);
   break;
case EVENT_KEYBOARD:
    keyboardEvent= (KEYSTROKE *) event;
    /*

* See if it's an Alt key
```

```
/* * OK, which Alt key.
            switch (keyboardEvent->scan) {
            case X_KEY:
               /*
* Return TRUE. This is the indication that we want to
                 * terminate.
                return TRUE;
                break;
            case U_KEY:
                /* 
* Change the attribute and update the cursor
                attrib++;
                currCell[0]= key;
                currCell[1]= attrib;
                VioWrtCellStr(currCell, 2, row, col, 0);
                break;
            case D KEY:
                /*

* Change the attribute and update the cursor
                attrib";
                currCell[0]= key;
                currCell[1]= attrib;
                VioWrtCellStr(currCell, 2, row, col, 0);
                break:
            default:
                /*
* Beep on an invalid Alt key
                DosBeep(100, 100);
                break;
            }
        else {
             * It's not an Alt key. Assume it's a standard ASCII key.
            key= keyboardEvent->ascii;
            currCell[0]= key;
            currCell[1]= attrib;
            VioWrtCellStr(currCell, 2, row, col, 0);
        break;
    default:
        ^{\prime*} ^{\star} A little sanity clause. (Everybody knows there ain't no
         * sanity clause :-)
        printf("Unknown event #%d\n", eventType);
        break;
    return FALSE;
}
```

if (keyboardEvent->shiftKeys & KBDSTF_ALT) {

Figure 15.1 Continued

```
keyboardThread()
        This procedure waits for keystrokes. When it gets a
   keystroke, it adds a keystroke event to the input queue.
void keyboardThread(void *foo)
{
   HKBD
                        keyboard;
   KBDKEYINFO
                        keyInfo;
                       *newKeystroke;
   KEYSTROKE
       Get access to the keyboard.
    KbdOpen(&keyboard);
    KbdGetFocus(IO_WAIT, keyboard);
       Loop waiting on the keyboard and queueing events.
   while (TRUE) {
       /*

* Wait for a keystroke
        KbdCharIn(&keyInfo, IO_WAIT, keyboard);
         * Add a request to the queue.
        newKeystroke= malloc(sizeof(KEYSTROKE));
        newKeystroke->ascii= keyInfo.chChar;
        newKeystroke->scan= keyInfo.chScan;
        newKeystroke->shiftKeys= keyInfo.fsState;
        DosWriteQueue(
                    inputQ,
                                            /* Queue handle
                    EVENT_KEYBOARD,
                                            /* Request ID: keyboard
                                            /* Length of data
                    sizeof(KEYSTROKE),
                                            /* Address of data
                    newKeystroke,
                                            /* Priority
                    0);
        }
}
        mouseThread()
        MouseThread waits for a mouse event and queues it to the
    input event queue.
void mouseThread(void *foo)
{
```

```
HMOU
                    mouHand;
MOUEVENTINFO
                 mouEvent;
USHORT
                   fWait;
USHORT
                   eventMask;
MOUSEEVENT
                   *newMouseEvent:
PTRLOC
                   location;
* Make the mouse available
MouOpen(NULL, &mouHand);
eventMask= MOUSE_BN1_DOWN | MOUSE_BN2_DOWN
            ! MOUSE_MOTION_WITH_BN1_DOWN
            ! MOUSE_MOTION_WITH_BN2_DOWN
            ! MOUSE_MOTION;
MouSetEventMask(&eventMask, mouHand);
* Generate a dummy event to give initial mouse location
MouGetPtrPos(&location, mouHand);
newMouseEvent= malloc(sizeof(MOUSEEVENT));
newMouseEvent->row= location.row;
newMouseEvent->col= location.col;
newMouseEvent->state= MOUSE MOTION;
/*

* Write it to the input queue
DosWriteQueue(
                                       /* Queue handle
                inputQ.
                                       /* Request ID: keyboard
                EVENT_MOUSE,
                                       /* Length of data
/* Address of data
/* Priority
                sizeof (MOUSEEVENT),
                newMouseEvent,
                0);
* Loop reading from the mouse queue and adding events to the input
* event queue.
while (TRUE) {
    * Wait for a mouse event
    fWait= 1;
   MouReadEventQue(&mouEvent, &fWait, mouHand):
    if (mouEvent.time) {
       /*
* Create a MOUSEEVENT
        newMouseEvent= malloc(sizeof(MOUSEEVENT));
        newMouseEvent->row= mouEvent.row;
        newMouseEvent->col= mouEvent.col;
        newMouseEvent->state= mouEvent.fs:
        * Add it to the event queue
        DosWriteQueue(
                inputQ.
                                       /* Queue handle
                                                                     */
```

Figure 15.1 Continued

```
/* Request ID: mouse
/* Length of data
/* Address of data
                            EVENT MOUSE,
                            sizeof(MOUSEEVENT),
                            newMouseEvent,
                                                              /* Priority
                 }
           }
}
```

This behavior is generated solely by the processEvent () procedure. The rest of the program is fairly generic. Virtually any application that uses the mouse and keyboard could be implemented by replacing the processEvent() procedure.

While this is a toy example, one can see that this technique could be used to develop powerful user-interface tools.

Summary

We have presented a simple application of queues, which illustrates how multiple streams of input can be merged into a single time-sequenced input stream.

Multithreading and the Presentation Manager

Chapter

16

Using Server Threads under PM

The Presentation Manager, OS/2's windowing system, is message based. Much like other windowing systems, programs are written to accept messages that describe events upon which action might or might not need to be taken. Some common events are: keystrokes, mouse movement, mouse button clicks, window repaints, window resizes, etc. Generally, a PM application will do all of its work in response to messages. Thus, a PM application is essentially a message processing loop. Specific applications are defined primarily in terms of how they respond to messages.

This approach is reasonably easily understood and encourages the design of user-centered applications: the user runs the program, the program doesn't run the user. The program simply reacts to the user's actions, rather than directing them.

One feature of this model is that a single-threaded application handles messages one at a time. This is good in that it makes application programming easier, but it has a major drawback when the processing required to service a message becomes lengthy.

To illustrate the problem that we are talking about, we have prog16-1.c, which is presented in Figure 16.1. This program requires two support files: an include file, prog16.1.h, which is presented in Figure 16.2, and a resource file, prog16-1-rc, which is presented in Figure 16.3. This program plays a simple game of Tic-Tac-Toe. Try playing a game or two. The interface is fairly standard and intuitive. One thing that you will notice is that while the computer is thinking the system doesn't respond to input. Note that if you have the clock or pulse running, you will see that the system isn't hung. You just can't do anything. Why is this?

Let's look a little bit at how the program is structured. PM applications respond to events. What events cause the computer to think? There are two cases in which the computer has to make a move. First, if the computer is starting the game, selecting gamernew will require the computer to make a move. Once play has begun, whenever the user makes a move, the computer must respond. To stack the deck against this program, we have used a tragically brute-force, exhaustive game tree search to find

a move. This takes a lot of time. This code is executed as part of servicing the messages that trigger it. Recall that a single-threaded PM application can service only one request at a time. Clearly, while it is thinking about its next move, it will not be able to respond to any user input. Thus, the application is incommunicado.

Figure 16.1 prog16-1.c

```
prog16-1.c
        This program is an example of the need for multithreading
    in presentation manager applications.
        It plays a simple game of tic-tac-toe. To choose a move,
    the program does an exhaustive search to the conclusion of the
    game. This takes a fair amount of time. If we do this
    computation while processing a window message, the whole work-
    place will be hung during the computation.
#include <stdlib.h>
#include <string.h>
#include <errno.h>
#define INCL_WIN
#define INCL_GPI
#include <os2.h>
#include "prog16-1.h"
 * Define the basic PM stuff:
HAB
                    anchorBlock;
HWND
                    ticTacToeWind;
HWND
                    frameWind;
 * Define the TicTacToe board.
TicTacToe
                        board[3][3];
                                           /* The board
                                                                    */
void main(void)
{
   HMQ
                        messageQ;
    OMSG
                        message;
   ULONG
                        createFlags;
       Get PM-ready
    anchorBlock= WinInitialize(0);
    if (anchorBlock == 0) {
        return;
    messageQ= WinCreateMsgQueue(anchorBlock, 0);
```

```
* Create our TicTacToe window class
   WinRegisterClass(
                       anchorBlock,
                                          /* Anchor Block
                                          /* Class Name
                       TICTACTOE,
                                          /* Window proc for class*/
                       ticTacToeWP.
                                          /* Redraw on a resize */
                       CS_SIZEREDRAW
                          CS_SYNCPAINT, /* Paint synchronously
                                          /* No user data
   createFlags=
           FCF_TITLEBAR
                           FCF SYSMENU
                                                      FCF_SIZEBORDER
           FCF_MINMAX
                          | FCF_SHELLPOSITION
                                                      FCF_TASKLIST
           FCF MENU;
       Create the window
   frameWind= WinCreateStdWindow(
                       HWND DESKTOP.
                                          /* Parent is desk top
                       WS_VISIBLE,
                                          /* Make frame visible
                                          /* Various flags
                       &createFlags,
                                          /* Window Class
                       TICTACTOE,
                       "Prog16-1 - Tic Tac Toe (single threaded)",
                                          /* Title
                                          /* Make client visible
                       WS_VISIBLE,
                                          /* Resources in .EXE
                       NULLHANDLE,
                                          /* Window ID
                       ID_MENU_MAIN,
                                          /* Client HWND
                       &ticTacToeWind);
   if (!frameWind) {
       WinPostMsg(NULLHANDLE, WM_QUIT, MPFROMSHORT(0));
       return;
       }
       Bring it up front and activate it.
   WinSetFocus(HWND_DESKTOP, ticTacToeWind);
       Start the message loop
   while (WinGetMsg(anchorBlock, &message, (HWND) NULL, 0, 0)) {
       WinDispatchMsg(anchorBlock, &message);
   WinDestroyWindow(frameWind);
   WinDestroyMsgQueue(messageQ);
   WinTerminate(anchorBlock);
}
       ticTacToeWP()
       Window procedure for the tic-tac-toe window.
```

Figure 16.1 Continued

```
static POINTL windowSize;
static POINTL boardCenter;
static long boardSize;
static TicTacToe nextMove= X:
static TicTacToe winner= MT;
static TicTacToe human;
static short
                    humanStart;
short
                    i:
short
                    j;
HPS
                    presSpace;
POINTL
                    location;
POINTL
                    boardUlc;
short
                    row;
short
                    col;
HWND
                    menuWind;
switch (msgNum) {
case WM COMMAND:
    switch (SHORT1FROMMP(msgParm1)) {
    case ID_MENU_HUMAN:
        /**

* Human Starts menu choice
        menuWind= WinWindowFromID(frameWind, FID_MENU);
        WinCheckMenuItem(menuWind, ID_MENU_HUMAN, TRUE);
        WinCheckMenuItem(menuWind, ID_MENU_COMPUTER, FALSE);
        humanStart= TRUE;
        break;
    case ID_MENU_COMPUTER:
        /*
* Computer Starts menu choice
        menuWind= WinWindowFromID(frameWind, FID MENU);
        WinCheckMenuItem(menuWind, ID_MENU_HUMAN, FALSE);
        WinCheckMenuItem(menuWind, ID_MENU_COMPUTER, TRUE);
        humanStart= FALSE;
        break;
    case ID_MENU_NEW:
            New Game menu choice
         * Reset to beginning:
        winner= MT;
        nextMove= X;
        if (humanStart) {
            human= X:
            }
        else {
            human= 0;
            Wipe the board
        for (row= 0; row < 3; row++) {
```

```
for (col = 0; col < 3; col ++) {
                board[row][col] = MT;
            }
           Clear the window
        presSpace= WinGetPS(window);
        GpiErase(presSpace);
        WinReleasePS(presSpace);
         * Redraw the board.
        WinInvalidateRect(window, NULL, FALSE);
           If the computer is starting, do it.
        if (nextMove != human) {
           computerMove(nextMove);
        break;
   break;
case WM_TICTACTOE_MOVE:
       We've gotten a message to perform a move. This could
    * be either from the computerMove procedure or from
    * a user button press.
    ^{\star} Put the move on the board.
   board[SHORT1FROMMP(msgParm2)][SHORT2FROMMP(msgParm2)]=
                                                 (TicTacToe) msgParm1;
   /*
 * See if there's a winner.
 */
   winner= ticTacToeThreeInARow(board);
    * Redraw the board with the new state
   WinInvalidateRect(window, NULL, FALSE);
    if (!winner) {
        * Game's not over. Go to next move.
        nextMove= OPPOSITE((TicTacToe) msgParm1);
   else {
        * Game's over. No next move.
        nextMove= MT;
    if (human == (TicTacToe) msgParm1) {
        * The computer might need to make a move
```

Figure 16.1 Continued

```
if (!winner) {
            ^{\prime*} ^{\star} The game's not over. Computer needs to generate a move.
            computerMove(nextMove);
        }
    break;
case WM_BUTTON1CLICK:
    /*
    * The human clicked. First see if it was his turn:
    if (nextMove != human) {
         * Bad human. Bad human. Naughty, Naughty.
        DosBeep(100, 50);
        break;
        }
        OK, now figure out where he clicked:
    location.x= SHORT1FROMMP(msgParm1);
    location.y= SHORT2FROMMP(msgParm1);
    boardUlc.x= boardCenter.x - boardSize / 2;
    boardUlc.y= boardCenter.y + boardSize / 2;
    row= boardUlc.y - location.y;
    col= location.x - boardUlc.x;
    ^{\prime*} ^{\star} Was it on the board area ?
    if (row < 0 !! row > boardSize !! col < 0 !! col > boardSize) {
        /*
* Bad human. Bad human. Naughty, Naughty.
        DosBeep(100, 50);
        break;
       Translate from pixel to row and col.
    row/= boardSize / 3;
    col/= boardSize / 3;
       Is the square empty ?
    if (board[row][col] != MT) {
        /*

* Bad human. Bad human. Naughty, Naughty.
        DosBeep(100, 50);
        break;
        }
```

```
* Send off the move.
    WinSendMsg(window, WM_TICTACTOE_MOVE, (MPARAM) nextMove,
                                        MPFROM2SHORT(row, col));
   break;
case WM_CREATE:
   /*

* Set up the default to human starts.
   menuWind= WinWindowFromID(WinQueryWindow(window, QW_PARENT),
                                FID MENU);
    WinCheckMenuItem(menuWind, ID_MENU_HUMAN, TRUE);
    humanStart= TRUE;
    nextMove= X;
    human= X;
    * Clear the board.
    for (row = 0; row < 3; row ++) {
        for (col= 0; col < 3; col++) {
           board[row][col]= MT;
    break;
case WM_SIZE:
    * Recalculate the size of the board. Leave MESSAGE_ALLOWANCE
    ^{\ast} pixels at the top of the window for the win, lose, draw message.
    windowSize.x= SHORT1FROMMP(msgParm2);
    windowSize.v= SHORT2FROMMP(msqParm2) - MESSAGE ALLOWANCE:
    * Make board 85% of the window size.
    * /
    boardSize= 85 * min(windowSize.x, windowSize.y);
    boardSize/= 100;
    boardCenter.x= windowSize.x / 2;
    boardCenter.y= windowSize.y / 2;
   break;
case WM_PAINT:
   /*
* Draw the board.
    presSpace= WinBeginPaint(window, NULLHANDLE, NULL);
    * Draw vertical lines
    location.x= boardCenter.x - boardSize / 6;
    location.y= boardCenter.y + boardSize / 2;
   GpiMove(presSpace, &location);
    location.y= boardCenter.y - boardSize / 2;
   GpiLine(presSpace, &location);
    location.x= boardCenter.x + boardSize / 6;
    location.y= boardCenter.y + boardSize / 2;
    GpiMove(presSpace, &location);
    location.y= boardCenter.y - boardSize / 2;
```

Figure 16.1 Continued

```
GpiLine(presSpace, &location);
   Draw horizontal lines
location.x= boardCenter.x - boardSize / 2;
location.y= boardCenter.y + boardSize / 6;
GpiMove(presSpace, &location);
location.x= boardCenter.x + boardSize / 2;
GpiLine(presSpace, &location);
location.x= boardCenter.x - boardSize / 2;
location.y= boardCenter.y - boardSize / 6;
GpiMove(presSpace, &location);
location.x= boardCenter.x + boardSize / 2;
GpiLine(presSpace, &location);
   Now draw any marks that there might be
for (i = 0; i < 3; i++) {
    for (j = 0; j < 3; j++) {
        switch (board[i][j]) {
        case MT:
            break:
        case 0:
            location.x= boardCenter.x + (j - 1) * boardSize / 3;
            location.y= boardCenter.y - (i - 1) * boardSize / 3;
            GpiMove(presSpace, &location);
            GpiFullArc(presSpace, DRO_OUTLINE,
                        MAKEFIXED(boardSize / 8, 0));
            break:
        case X:
            location.x= boardCenter.x + (j - 1) * boardSize / 3
                            - boardSize / 8;
            location.y= boardCenter.y - (i - 1) * boardSize / 3
                            + boardSize / 8;
            GpiMove(presSpace, &location);
            location.x+= boardSize * 2 / 8;
            location.y-= boardSize * 2 / 8;
            GpiLine(presSpace, &location);
            location.x= boardCenter.x + (i - 1) * boardSize / 3
                            + boardSize / 8;
            location.y= boardCenter.y - (i - 1) * boardSize / 3
                            + boardSize / 8;
            GpiMove(presSpace, &location);
            location.x-= boardSize * 2 / 8;
            location.y-= boardSize * 2 / 8;
            GpiLine(presSpace, &location);
            break;
    }
^{\prime\ast} ^{} If the game has ended, put out an appropriate message.
if (winner) {
    char
               *message;
    POINTL
                txtBox[TXTBOX_COUNT];
```

```
if (winner == DRAW) {
                message= "Ho hum, another draw";
            else if (winner == human) {
                message= "Congratulations, you won !";
            else {
                message= "You lost. How do you feel about that ?";
            {\tt GpiQueryTextBox(presSpace, strlen(message), message,}
                                     TXTBOX_COUNT, txtBox);
            location.x= (windowSize.x - txtBox[TXTBOX_BOTTOMRIGHT].x
                                      + txtBox[TXTBOX_BOTTOMLEFT].x) / 2;
            location.y= windowSize.y + (MESSAGE_ALLOWANCE
                                         - txtBox[TXTBOX_TOPLEFT].y
                                         + txtBox[TXTBOX_BOTTOMLEFT].y) / 2;
            GpiCharStringAt(presSpace, &location, strlen(message), message);
        WinEndPaint(presSpace);
        break;
    case WM_ERASEBACKGROUND:
        return (MRESULT) TRUE;
        break:
    default:
        return WinDefWindowProc(window, msgNum,
                    msgParm1, msgParm2);
    return FALSE;
}
        computerMove()
        This function finds a move for the computer. It then sends a
    WM_TICTACTOE_MOVE message to the window.
void
        computerMove(TicTacToe player)
    TicTacToe
                        localBoard[3][3];
    short
                        row, col;
        Set the "Wait" pointer. It could be a _long_ wait.
   WinSetPointer(
                        HWND_DESKTOP,
                        WinQuerySysPointer(
                            HWND_DESKTOP,
                            SPTR_WAIT,
                            TRUE));
       Create a local copy of the board:
   memcpy(localBoard, board, sizeof(board));
```

Figure 16.1 Continued

```
Find the best move.
   pickMove(localBoard, player, OPPOSITE(player), &row, &col);
    * Now send a message for the desired move.
   WinSendMsg(ticTacToeWind, WM_TICTACTOE_MOVE, (MPARAM) player,
                                MPFROM2SHORT(row, col));
       Set the normal pointer
   WinSetPointer(
                        HWND_DESKTOP,
                        WinQuerySysPointer(
                            HWND_DESKTOP,
                            SPTR_ARROW,
                            TRUE));
}
        pickMove()
        This procedure evaluates the position. It returns the score
   for the current position, and if desired, the suggested move.
   The score is coded as follows:
        - 2
                player has already lost
        - 1
                player will lose
        0
                game will draw
         1
                player will win
                player has already won
   If moveRow and moveCol are non-NULL pointers, the suggested move
    will be returned through them.
short pickMove(TicTacToe board[3][3],
                    TicTacToe player, TicTacToe opponent,
                    short *moveRow, short *moveCol)
    short
                        row, col, score;
                        bestRow, bestCol, bestScore;
    short
    TicTacToe
                        winner;
    ^{\star} See whether someone's already won.
    winner= ticTacToeThreeInARow(board);
    if (winner == player) {
        return 2;
    else if (winner == opponent) {
        return -2;
    else if (winner == DRAW) {
        return 0;
```

}

```
Now evaluate possible moves.
        Initialize bestScore to an impossibly bad score. Any legal move
        will be better. If there were no legal moves, we would have found
        out above.
    bestScore= -3;
    for (row= 0; row < 3; row++) {
        for (col = 0; col < 3; col ++) {
            if (board[row][col] == MT) {
                /*
 * The move is legal. Evaluate it.
                board[row][col]= player;
                    Note that we've reversed the opponent and player; the
                 * next move is the opponents. Note also the negative
                 ^{\star} sign, what's good for the opponent is bad for the
                    player, and vice-versa.
                score= -pickMove(board, opponent, player, NULL, NULL);
                 * Erase the move we just evaluated.
                board[row][col]= MT;
                if (score > bestScore) {
                    bestScore= score;
                    bestRow= row;
                    bestCol= col;
                    }
                }
            }
        }
       Return the suggested move if the caller requested it.
    if (moveRow) {
        *moveRow= bestRow;
    if (moveCol) {
        *moveCol= bestCol;
    if (bestScore < 0) {
        return -1;
    else if (bestScore == 0) {
       return O:
    else {
        return 1;
        }
}
```

Figure 16.1 Continued

```
ticTacToeThreeInARow()
        This function checks the board for a winner and returns the
    winner or MT (0) if there is no winner.
TicTacToe
            ticTacToeThreeInARow(TicTacToe board[3][3])
{
    short
                         i;
    TicTacToe
                         side;
    short
                         row, col;
        Win is an array of the eight possible "rows" that comprise a win.
    static struct {
                             row, col;
             short
                         win[8][3] =
             {{{0,0}, {0,1}, {0,2}},
              \{\{1,0\}, \{1,1\}, \{1,2\}\},\
              {{2,0}, {2,1}, {2,2}}, {{0,0}, {1,0}, {2,0}},
              {{0,1}, {1,1}, {2,1}},
              {{0,2}, {1,2}, {2,2}},
              {{0,0}, {1,1}, {2,2}},
              {{0,2}, {1,1}, {2,0}}};
        See if someone's won.
     */
    for (side= X; side <= 0; side++) {
        for (i = 0; i < 8; i++) {
             if (board[win[i][0].row][win[i][0].col] == side &&
                     board[win[i][1].row][win[i][1].col] == side &&
                     board[win[i][2].row][win[i][2].col] == side) {
                 break;
                 }
             }
        if (i < 8) {
             return side;
        }
        No one's won. See whether it's a draw.
    for (row = 0; row < 3; row ++) {
        for (col = 0; col < 3; col ++) {
             if (board[row][col] == MT) {
                    Not a draw, return MT.
                 return MT;
        }
    return DRAW;
}
```

Figure 16.2 prog16-1.h

```
prog16-1.h
        Generally include file for TicTacToe examples.
   Define the IDs of the pointer shapes:
#define CLOCKO_ID
                            1000
                            1001
#define CLOCK1_ID
#define CLOCK2_ID
                            1002
#define CLOCK3_ID
                            1003
#define CLOCK4_ID
                            1004
#define CLOCK5_ID
                            1005
#define CLOCK6_ID
                            1006
#define CLOCK7_ID
                            1007
* Define the menu element IDs
#define ID_MENU_MAIN
#define ID MENU GAME
                            210
#define ID_MENU_NEW
                            211
#define ID_MENU_OPTIONS
                            220
#define ID_MENU_HUMAN
                            221
#define ID_MENU_COMPUTER
                            222
/*

* Define the TicTacToe window class
#define TICTACTOE
                        "TicTacToe"
 * Define the TicTacToe window ID
#define WID_TICTACTOE
   Define the number of pixels of height we're going to allow for
   the win, lose, draw message.
#define MESSAGE ALLOWANCE 30
/*
* Define all of the user messages
#define WM_TICTACTOE_MOVE
                          (WM_USER+100)
 * Define the basic TicTacToe data type:
typedef enum {
   MT=
                0x0,
   X=
                0x10,
   0=
                0x11.
   DRAW=
                0x20
   } TicTacToe;
/*
```

Figure 16.2 Continued

```
* Define a macro to give the opposing mark.
#define OPPOSITE(x)
                       ((TicTacToe) ((short)(x) ^ 1))
   Define window procedure for TicTacToe class
FNWP
                        ticTacToeWP;
   Define miscellaneous procedures
*/
                        computerMove(TicTacToe plauer);
void
void
                        computerMoveMT(HWND window, TicTacToe plauer);
void
                        moveEvalThread(void *semaphore);
                        windowError(HWND window, char *text, ULONG error);
void
                        pickMove(TicTacToe board[3][3],
short
                                    TicTacToe player, TicTacToe opponent,
                                    short *row, short *col);
TicTacToe
                        ticTacToeThreeInARow(TicTacToe board[3][3]);
```

Figure 16.3 prog16-1.rc

This is fine, because there isn't much you would want to do, except perhaps abort the game. If you try to access another application, however, you will find that you really can't do anything at all. It is somewhat less clear why this is.

In the design of a windowing system, there is a choice that has to be made. There is no "right" choice. Both alternatives have drawbacks. The choice is this: should user input be serialized or not? In OS/2, input is serialized. An input event is not dispatched (sent to a window message queue) until the previous input event has been serviced. This goes for the system as a whole. Thus, because the Tic-Tac-Toe program is sitting in a message handler figuring out its next move, no other input events will be serviced. Non-input events in other processes do get serviced. This is why the clock and pulse, for example, keep on running.

Why design a system this way? The alternative is not to explicitly serialize input events. This choice has drawbacks as well. Think about this operation: click on a window to activate it, and start typing. When input is serialized, the typing will always go to the new window. As part of servicing the click, the new window will be made active, then, when the keystrokes are handled, they will be directed to the correct

window. Without explicit serialization, you might find that, after you click on a window, keystrokes keep going to the previously active window for a short time before the new window actually is made active. This is particularly noticeable under heavy load. This is because the keystrokes are directed to the active window before the processing of the click is complete. It is somewhere in the click processing that the active window changes. It is at this point that the keystrokes start going to the new window.

The good news is this: if the windowing system that you use does not serialize input, there's nothing you as an application developer can do about it but live with it. When input is serialized, as in OS/2, you can avoid some of the drawbacks. The solution is, of course, multithreading.

In prog16-2.c, which is presented in Figure 16.4, we present a multithreading solution. This program uses the same include file as prog16-1, prog16-1.h, and its own resource file, prog16-2.rc, which is presented in Figure 16.5. A server thread is created to do Tic-Tac-Toe move evaluation. When it becomes necessary for the computer to make a move, the server thread is signalled to find a move.

Figure 16.4 prog16-2.c

```
prog16-2.c
        This program solves the problems in prog16-1 through
    multithreading.
        In this program, we create a server thread that processes
    requests for move evaluation. This frees the window thread to
    service messages and so allows the PM to get input.
#include <stdlib.h>
#include <string.h>
#include <errno.h>
#include cess.h>
#define INCL_WIN
#define INCL_GPI
#define INCL_DOSPROCESS
#define INCL DOSSEMAPHORES
#include <os2.h>
#include "mt.h"
#include "prog16-1.h"
    Define the basic PM stuff:
HAB
                    anchorBlock;
HWND
                    ticTacToeWind;
HWND
                    frameWind;
    Define the TicTacToe board.
TicTacToe
                        board[3][3];
                                            /* The board
```

Figure 16.4 Continued

```
* Set up a variable to communicate to the server thread
TicTacToe
                       evalRequest= MT;
* The semaphore to poke server
HEV
                       evalRequestSem;
* Define some stuff for making a nice wait cursor
HPOINTER
            currPointer;
HPOINTER
           clockPtr[8];
long
            clockTime;
       main()
void main(void)
{
   HMQ
                      messageQ;
   OMSG
                      message;
   ULONG
                      createFlags;
   USHORT
                      serverTid;
    * Get PM-ready
    anchorBlock= WinInitialize(0);
    if (anchorBlock == 0) {
       return;
       }
   messageQ= WinCreateMsgQueue(anchorBlock, 0);
    * Create our TicTacToe window class
   WinRegisterClass(
                       anchorBlock,
                                          /* Anchor Block
                                         /* Class Name
                       TICTACTOE,
                                        /* Window proc for class*/
                       ticTacToeWP,
                                         /* Redraw on a resize */
                       CS_SIZEREDRAW
                          CS_SYNCPAINT, /* Paint syncronously
                                          /* No user data
    createFlags=
           FCF_TITLEBAR
                          FCF_SYSMENU
                                           FCF_SIZEBORDER
                          | FCF_SHELLPOSITION | FCF_TASKLIST
           FCF_MINMAX
           FCF_MENU;
       Create the window
    frameWind= WinCreateStdWindow(
                                         /* Parent is desk top */
                       HWND_DESKTOP,
                                         /* Make frame visible */
                       WS_VISIBLE,
                                          /* Various flags
                       &createFlags,
```

```
/* Window Class
                        TICTACTOE,
                        "Prog16-2 - Tic Tac Toe (multi-threaded)",
                                          /* Title
                        WS VISIBLE.
                                            /* Make client visible
                        NULLHANDLE,
                                            /* Resources in .EXE
                                           /* Window ID
                        ID_MENU_MAIN,
                        &ticTacToeWind);
                                           /* Client HWND
        Bring it up front and activate it.
    WinSetFocus(HWND_DESKTOP, ticTacToeWind);
     ^{\star} Create the event semaphore that the server thread will wait
     * on. We'll post it when we need service.
    DosCreateEventSem(
                        NULL,
                                            /* Unnamed semaphore
                        &evalRequestSem,
                                           /* Handle
                                            /* No options not shared*/
                        FALSE);
                                            /* Initially reset
    if (!frameWind) {
        WinPostMsg(NULLHANDLE, WM_QUIT, MPFROMSHORT(0), MPFROMSHORT(0));
        Start the server thread
    serverTid= _beginthread(
                        moveEvalThread,
                                            /* Start address
                                            /* Dummmy stack addr
                        NULL,
                        STACK_SIZE,
                                            /* Stack size
                        NULL);
                                            /* No message
       Start the message loop
    while (WinGetMsg(anchorBlock, &message, (HWND) NULL, 0, 0)) {
        WinDispatchMsg(anchorBlock, &message);
    WinDestroyWindow(frameWind);
    WinDestroyMsgQueue(messageQ);
    WinTerminate(anchorBlock):
}
        ticTacToeWP()
        Window procedure for the tic tac toe window.
MRESULT EXPENTRY ticTacToeWP(HWND window, ULONG msgNum,
           MPARAM msgParm1, MPARAM msgParm2)
{
    static POINTL
                        windowSize:
    static POINTL
                        boardCenter:
```

Figure 16.4 Continued

```
static long
                    boardSize;
static TicTacToe
                    nextMove;
static TicTacToe
                    winner;
static TicTacToe
                    human;
static short
                    humanStart;
static HPOINTER
                    oldPointer:
short
                    i:
short
                    i;
                    presSpace:
HPS
POINTL
                    location;
POINTL
                    boardUlc;
short
                    IOW;
short
                    col;
HWND
                   menuWind;
switch (msgNum) {
case WM COMMAND:
    switch (SHORT1FROMMP(msgParm1)) {
    case ID_MENU_HUMAN:
        * Human Starts menu choice
         */
        menuWind= WinWindowFromID(frameWind, FID_MENU);
        WinCheckMenuItem(menuWind, ID_MENU_HUMAN, TRUE);
        WinCheckMenuItem(menuWind, ID_MENU_COMPUTER, FALSE);
        humanStart= TRUE;
        break;
    case ID_MENU_COMPUTER:
        * Computer Starts menu choice
        menuWind= WinWindowFromID(frameWind, FID_MENU);
        WinCheckMenuItem(menuWind, ID_MENU_HUMAN, FALSE);
        WinCheckMenuItem(menuWind, ID_MENU_COMPUTER, TRUE);
        humanStart= FALSE;
        break;
    case ID_MENU_NEW:
         * New Game menu choice
         * Reset to beginning:
         */
        winner= MT;
        nextMove= X;
        if (humanStart) {
            human= X;
        else {
            human= 0;
            }
           Wipe the board
        for (row= 0; row < 3; row++) {
            for (col= 0; col < 3; col++) {
                board[row][col]= MT;
```

```
}
            }
        /*
* Clear the window
        presSpace= WinGetPS(window);
        GpiErase(presSpace);
        WinReleasePS(presSpace);
         * Redraw the board.
        WinInvalidateRect(window, NULL, FALSE);
            If the computer is starting, start the clock and poke
            the server.
        if (nextMove != human) {
            computerMoveMT(window, nextMove);
    break;
case WM_TICTACTOE_MOVE:
   /*

* We've gotten a message to perform a move. This could
     * be either from the server task or from a user button press.
    board[SHORT1FROMMP(msgParm2)][SHORT2FROMMP(msgParm2)]=
                                                 (TicTacToe) msgParm1;
     * See if there's a winner.
    winner= ticTacToeThreeInARow(board);
    * Redraw the board with the new state
    WinInvalidateRect(window, NULL, FALSE);
    if (!winner) {
        /*
* Game's not over. Go to next move
        nextMove= OPPOSITE((TicTacToe) msgParm1);
    else {
        * Game's over. No next move.
        nextMove= MT;
    if (human == (TicTacToe) msgParm1) {
        /*
    * The computer might need to make a move
        if (!winner) {
            ^{\prime*} ^{} If this was a human move, then do a computer move
            computerMoveMT(window, nextMove);
```

Figure 16.4 Continued

```
}
   else {
           This was a computer move. Return to the normal pointer.
       WinStopTimer(anchorBlock, window, 0);
        currPointer= oldPointer;
        WinSetPointer(HWND_DESKTOP, currPointer);
            Re-enable the menu options
       menuWind= WinWindowFromID(frameWind, FID_MENU);
        WinEnableMenuItem(menuWind, ID_MENU_GAME, TRUE);
       WinEnableMenuItem(menuWind, ID_MENU_OPTIONS, TRUE);
        }
    break;
case WM_BUTTON1CLICK:
   /*
    * The human clicked. First see if it was his turn:
    if (nextMove != human) {
        /*

* Bad human. Bad human. Naughty, Naughty.
        DosBeep(100, 50);
        break;
   /*
 * OK, now figure out where he clicked:
    location.x= SHORT1FROMMP(msgParm1);
    location.y= SHORT2FROMMP(msgParm1);
    boardUlc.x= boardCenter.x - boardSize / 2;
    boardUlc.y= boardCenter.y + boardSize / 2;
    row= boardUlc.y - location.y;
    col= location.x - boardUlc.x;
       Was it on the board area?
    if (row < 0 !! row > boardSize !! col < 0 !! col > boardSize) {
        /*

* Bad human. Bad human. Naughty, Naughty.
        DosBeep(100, 50);
        break;
     * Translate from pixel to row and col.
    row/= boardSize / 3;
    col/= boardSize / 3;
        Is the square empty?
```

```
if (board[row][col] != MT) {
               Bad human. Bad human. Naughty, Naughty.
          DosBeep(100, 50):
          break;
          Send off the move.
     WinSendMsg(window, WM_TICTACTOE_MOVE, (MPARAM) nextMove,
                                                  MPFROM2SHORT(row, col));
     break;
case WM CREATE:
      * Get the standard pointer.
     oldPointer= WinQuerySysPointer(HWND_DESKTOP, SPTR ARROW, FALSE);
      * Get all of the sundial pointers.
     currPointer= oldPointer;
     clockPtr[0]= WinLoadPointer(HWND_DESKTOP, 0, CLOCK0_ID);
    clockPtr[0]= WinLoadPointer(HWND_DESKTOP, 0, CLOCK1_ID);
clockPtr[1]= WinLoadPointer(HWND_DESKTOP, 0, CLOCK1_ID);
clockPtr[2]= WinLoadPointer(HWND_DESKTOP, 0, CLOCK2_ID);
clockPtr[3]= WinLoadPointer(HWND_DESKTOP, 0, CLOCK3_ID);
clockPtr[4]= WinLoadPointer(HWND_DESKTOP, 0, CLOCK4_ID);
clockPtr[5]= WinLoadPointer(HWND_DESKTOP, 0, CLOCK5_ID);
     clockPtr[6] = WinLoadPointer(HWND_DESKTOP, 0, CLOCK6_ID);
     clockPtr[7] = WinLoadPointer(HWND_DESKTOP, 0, CLOCK7_ID);
      * Set up the default to human starts.
     menuWind= WinWindowFromID(WinQueryWindow(window, QW_PARENT),
                                        FID_MENU);
     WinCheckMenuItem(menuWind, ID MENU HUMAN, TRUE);
     humanStart= TRUE;
     nextMove= X;
     human= X;
      * Clear the board.
     for (row= 0; row < 3; row++) {
          for (col = 0; col < 3; col ++) {
               board[row][col]= MT;
     break;
case WM_SIZE:
     /*
         Recalculate the size of the board. Leave MESSAGE ALLOWANCE
         pixels at the top of the window for the win, lose, draw message.
     windowSize.x= SHORT1FROMMP(msgParm2);
     windowSize.y= SHORT2FROMMP(msgParm2) - MESSAGE_ALLOWANCE;
      * Make board 85% of the window size.
```

Figure 16.4 Continued

```
boardSize= 85 * min(windowSize.x, windowSize.y);
   boardSize/= 100;
   boardCenter.x= windowSize.x / 2;
   boardCenter.y= windowSize.y / 2;
   break;
case WM_TIMER:
   /*
       We're waiting for server task to pick a move. Update the
       clock pointer... BUT only if the pointer is over the
       client window.
   WinQueryPointerPos(HWND DESKTOP, &location):
    if (WinWindowFromPoint(HWND_DESKTOP, &location, TRUE) != window) {
       break;
    clockTime= (clockTime+1) % 8;
    currPointer= clockPtr[clockTime];
    WinSetPointer(HWND_DESKTOP, currPointer);
   break;
case WM_MOUSEMOVE:
   /* -
* Reset the pointer
    WinSetPointer(HWND_DESKTOP, currPointer);
    return (MRESULT) TRUE;
case WM_PAINT:
   /*
* Draw the board.
    presSpace= WinBeginPaint(window, NULLHANDLE, NULL);
     * Draw vertical lines
    location.x= boardCenter.x - boardSize / 6;
    location.y= boardCenter.y + boardSize / 2;
    GpiMove(presSpace, &location);
    location.y= boardCenter.y - boardSize / 2;
    GpiLine(presSpace, &location);
    location.x= boardCenter.x + boardSize / 6;
    location.y= boardCenter.y + boardSize / 2;
    GpiMove(presSpace, &location);
    location.y= boardCenter.y - boardSize / 2;
    GpiLine(presSpace, &location);
     * Draw horizontal lines
    location.x= boardCenter.x - boardSize / 2;
    location.y= boardCenter.y + boardSize / 6;
    GpiMove(presSpace, &location);
    location.x= boardCenter.x + boardSize / 2;
    GpiLine(presSpace, &location);
```

```
location.x= boardCenter.x - boardSize / 2;
location.y= boardCenter.y - boardSize / 6;
GpiMove(presSpace, &location);
location.x= boardCenter.x + boardSize / 2;
GpiLine(presSpace, &location);
* Now draw any marks that there might be
for (i = 0; i < 3; i++) {
    for (j = 0; j < 3; j++) {
        switch (board[i][j]) {
        case MT:
            break;
        case 0:
            location.x= boardCenter.x + (j - 1) * boardSize / 3;
            location.y= boardCenter.y - (i - 1) * boardSize / 3;
            GpiMove(presSpace, &location);
            GpiFullArc(presSpace, DRO_OUTLINE,
                        MAKEFIXED(boardSize / 8, 0));
            break:
        case X:
            location.x= boardCenter.x + (j - 1) * boardSize / 3
                            - boardSize / 8;
            location.y= boardCenter.y - (i - 1) * boardSize / 3
                            + boardSize / 8;
            GpiMove(presSpace, &location);
            location.x+= boardSize * 2 / 8;
            location.y-= boardSize * 2 / 8;
            GpiLine(presSpace, &location);
            location.x= boardCenter.x + (j - 1) * boardSize / 3
                            + boardSize / 8;
            location.y= boardCenter.y - (i - 1) * boardSize / 3
                            + boardSize / 8;
            GpiMove(presSpace, &location);
            location.x-= boardSize * 2 / 8;
location.y-= boardSize * 2 / 8;
            GpiLine(presSpace, &location);
            break;
        }
    }
   If the game has ended put out an appropriate message.
if (winner) {
   char
               *message;
   POINTL
                txtBox[TXTBOX_COUNT];
    if (winner == DRAW) {
       message= "Ho hum, another draw";
    else if (winner == human) {
       message= "Congratulations, you won !";
   else {
       message= "You lost. How do you feel about that ?";
   GpiQueryTextBox(presSpace, strlen(message), message,
```

Figure 16.4 Continued

```
TXTBOX_COUNT, txtBox);
            location.x= (windowSize.x - txtBox[TXTBOX_BOTTOMRIGHT].x
                                      + txtBox[TXTBOX_BOTTOMLEFT].x) / 2;
            location.y= windowSize.y + (MESSAGE_ALLOWANCE
                                        - txtBox[TXTBOX_TOPLEFT].y
                                        + txtBox[TXTBOX_BOTTOMLEFT].y) / 2;
            GpiCharStringAt(presSpace, &location, strlen(message), message);
        WinEndPaint(presSpace);
        break:
    case WM ERASEBACKGROUND:
        return (MRESULT) TRUE;
        break;
    default:
        return WinDefWindowProc(window, msgNum,
                   msgParm1, msgParm2);
    return FALSE;
        computerMoveMT()
        This function finds a move for the computer. It then sends a
    WM_TICTACTOE_MOVE message to the window.
        computerMoveMT(HWND window, TicTacToe player)
void
    HWND
                        menuWind;
        Set things up for our clock pointer
    clockTime= 0;
    currPointer= clockPtr[clockTime];
    WinSetPointer(HWND_DESKTOP, currPointer);
    WinStartTimer(anchorBlock, window, 0, 250);
        Disable the menu options. Changing things in the middle
        of a computer move would be unfortunate
    menuWind= WinWindowFromID(frameWind, FID_MENU);
    WinEnableMenuItem(menuWind, ID_MENU_GAME, FALSE);
    WinEnableMenuItem(menuWind, ID_MENU_OPTIONS, FALSE);
        And poke the server task to start formulating a response
    evalRequest= player;
    DosPostEventSem(evalRequestSem);
}
```

```
moveEvalThread()
        This procedure is invoked as a separate server thread. It
   waits on the evalRequestSem semaphore for move evaluation
   requests.
void moveEvalThread(void *_)
{
   TicTacToe
                        localBoard[3][3];
   TicTacToe
                        player;
   short
                        row, col;
   ULONG
                        postCount;
   while (TRUE) {
        ^{\prime*} ^{\ast} Wait on the semaphore till we get kicked
        DosWaitEventSem(
                        evalRequestSem,
                                                /* Semaphore handle
                                               /* Wait forever
                        SEM_INDEFINITE_WAIT);
           Reset the semaphore so we don't loop twice.
        DosResetEventSem(
                        evalRequestSem,
                                                /* Semaphore handle
                        &postCount);
                                                /* Better be 1
           Now pick up the request from the global location
       player= evalRequest;
          Create a local copy of the game board. We must do this. If
           we mess up the global board, and a repaint message comes, the
        * in-progress board will be displayed.
       memcpy(localBoard, board, sizeof(board));
        /*
* Find the best move.
        pickMove(localBoard, player, OPPOSITE(player), &row, &col);
        ^{\ast} \, Now POST a message for the desired move. Note that we
           are a different thread from the ticTacToeWind, Further,
           we are not PM-ready (i.e. we do not have a message queue).
           So we cannot SEND a message, we must POST it to the window's
          message queue.
       WinPostMsg(ticTacToeWind, WM_TICTACTOE_MOVE, (MPARAM) player,
                                MPFROM2SHORT(row, col));
       }
```

Figure 16.4 Continued

```
pickMove()
        This procedure evaluates the position. It returns the score
   for the current position and, if desired, the suggested move.
   The score is coded as follows:
        -2
               player has already lost
        - 1
                player will lose
        0
                game will draw
         1
                player will win
        2
                player has already won
   If moveRow and moveCol are non-NULL pointers, the suggested move
   will be returned through them.
short pickMove(TicTacToe board[3][3],
                    TicTacToe player, TicTacToe opponent,
                    short *moveRow, short *moveCol)
{
    short
                        row, col, score;
                        bestRow, bestCol, bestScore;
   short
   TicTacToe
                        winner;
       See whether someone's already won.
   winner= ticTacToeThreeInARow(board);
    if (winner == player) {
        return 2;
        }
    else if (winner == opponent) {
        return -2;
    else if (winner == DRAW) {
       return 0;
        }
       Now evaluate possible moves.
       Initialize bestScore to an impossibly bad score. Any legal move
       will be better. If there were no legal move, we would have found
       out above.
    bestScore= -3;
    for (row = 0; row < 3; row ++) {
        for (col= 0; col < 3; col++) {
            if (board[row][col] == MT) {
                    The move is legal. Evaluate it.
                board[row][col]= player;
                 * Note that we've reversed the opponent and player; the
                 * next move is the opponents. Note also the negative
                 \mbox{\ensuremath{\star}} sign, what's good for the opponent is bad for the
                    player, and vice-versa.
```

```
score= -pickMove(board, opponent, player, NULL, NULL);
                 * Erase the move we just evaluated.
                board[row][col]= MT;
                if (score > bestScore) {
                    bestScore= score;
                    bestRow= row;
                    bestCol= col;
            }
        }
        Return the suggested move if the caller requested it.
    if (moveRow) {
        *moveRow= bestRow;
    if (moveCol) {
        *moveCol= bestCol;
    if (bestScore < 0) {
        return -1;
    else if (bestScore == 0) {
        return 0;
        }
   else {
        return 1;
}
        ticTacToeThreeInARow()
        This function checks the board for a winner and returns the
    winner or MT (0) if there is no winner.
TicTacToe ticTacToeThreeInARow(TicTacToe board[3][3])
    short
                        i;
    TicTacToe
                        side:
   short
                        row, col;
    ^{\prime} win is an array of the eight possible "rows" that comprise a win.
    * /
   static struct {
            short
                           row, col;
                        win[8][3] =
            {{{0,0}, {0,1}, {0,2}},
             {{1,0}, {1,1}, {1,2}},
             \{\{2,0\}, \{2,1\}, \{2,2\}\},\
             {{0,0}, {1,0}, {2,0}},
```

Figure 16.4 Continued

```
{{0,1}, {1,1}, {2,1}},
             \{\{0,2\}, \{1,2\}, \{2,2\}\},\
             {{0,0}, {1,1}, {2,2}},
             \{\{0,2\}, \{1,1\}, \{2,0\}\}\};
        See if someone's won.
    for (side= X; side <= 0; side++) {
        for (i = 0; i < 8; i++) {
            if (board[win[i][0].row][win[i][0].col] == side &&
                    board[win[i][1].row][win[i][1].col] == side &&
                    board[win[i][2].row][win[i][2].col] == side) {
                break;
                }
        if (i < 8) {
            return side;
            }
        No one's won. See whether it's a draw.
    for (row= 0; row < 3; row++) {
        for (col= 0; col < 3; col++) {
            if (board[row][col] == MT) {
                    Not a draw, return MT.
                 * /
                return MT;
            }
    return DRAW;
Figure 16.5 prog16-2.rc
#include <os2.h>
#include "prog16-1.h"
MENU ID_MENU_MAIN {
    SUBMENU "~Game",
                                 ID_MENU_GAME {
        MENUITEM "~New",
                                 ID_MENU_NEW
    SUBMENU "~Options",
                                 ID_MENU_OPTIONS {
        MENUITEM "~Human Starts", ID_MENU_HUMAN
        MENUITEM "~Computer Starts", ID_MENU_COMPUTER
        }
    }
            CLOCKO ID
                             CLOCKO.PTR
pointer
pointer
            CLOCK1_ID
                             CLOCK1.PTR
            CLOCK2_ID
                             CLOCK2.PTR
pointer
                             CLOCK3.PTR
pointer
            CLOCK3_ID
                             CLOCK4.PTR
pointer
            CLOCK4_ID
                             CLOCK5.PTR
pointer
            CLOCK5_ID
            CLOCK6_ID
                             CLOCK6.PTR
pointer
```

CLOCK7.PTR

pointer

CLOCK7_ID

Before getting into the innards of prog16.2, it is necessary to discuss some of the details of the Presentation Manager as it relates to multithreading. In a PM application, there are two kinds of threads: message queue threads and non-message queue threads. Creation of a message queue via WinCreateMsgQueue() makes the calling thread a message queue thread. The thread remains a message queue thread until it calls WinDestroyMsgQueue() to delete the queue. When a message queue thread creates a window, the window becomes associated with that thread's message queue, and messages destined for that window will be serviced by that thread. In our program, there are two threads, the main thread and the server thread. The main thread is a message queue thread, and it services messages for the main window. The server thread is a nonmessage queue thread. Nonmessage queue threads cannot create windows or handle messages. They also cannot use WinSendMsg() or any call that sends rather than posts a message. Our server doesn't require any of these capabilities. More on this later.

The communication between the main thread and the server works as follows: the server thread is in a loop, waiting on a semaphore, then doing move evaluation. When the Tic-Tac-Toe window procedure needs the computer to make a move, it puts the computer's mark (X or O) into the global location evalRequest and posts the semaphore evalRequestSem. This wakes up the server thread, which then looks at evalRequest to see which mark it needs to find a move for. When it has decided upon a move, it posts a message back to the main window to make its move.

It is critical to understand the difference between posting and sending a message. Posting a message, via WinPostMsg(), adds a message to the target window's message queue and returns. The caller does not know when the message actually is processed. Sending a message, via WinSendMsg(), actually waits for the message to be processed. Only message queue threads are allowed to send messages. Any thread can post a message.

It is instructive to try making the server thread a message queue thread. If you add the following line to the beginning of the server thread, it will become a message-queue thread, allowing you to change the WinPostMsg() to a WinSendMsg():

```
WinCreateMsgQueue(WinInitialize(0), 0);
```

If you make this modification, you will find that, after the first move that the computer makes, the wait cursor slows down. This is due to the fact that the server thread gets a priority boost being a message-queue thread in the foreground process. Thus, the main thread, which services the timer messages that advance the wait cursor, no longer is getting priority service. Sometimes, it has to wait a while for the server threads time-slice to end.

Summary

Presentation Manager programs are messages based. Messages must be processed in a timely manner. If a PM application has a task to do that will require a significant amount of time, that task should be handed off to a server thread. The server thread can post a message back to the PM window with the results of its processing.



Chapter

17

Multithreaded Painting

In the previous chapter, we saw that sometimes lengthy processes need to be farmed off to server threads rather than being serviced by message queue threads to ensure good response time to window messages. There is one particular process that deserves special consideration: painting a window. Under the presentation manager, whenever a portion of a window goes from being obscured or invisible to being visible, a paint message is sent to the window procedure, telling it what area needs to be repainted. It is the window procedure's responsibility to paint that area of the window.

Typically, painting the window is a fairly quick procedure, and there is no difficulty doing the painting in the window procedure. However, what happens when painting is a slow process? There really are two solutions to this problem. The simplest is to actually generate the window image on a memory bitmap. Then, when a paint request is issued, the window procedure only needs to bitblt a portion of the memory image to the screen.

This is a common and reasonable approach. Its only drawback is that it can use quite a bit of memory, particularly at 256 or even 65,536 colors. Generally it's worth it, but there is another alternative. It is possible to shift the painting burden to a server thread, as we did for the Tic-Tac-Toe move computation in the previous chapter.

In this chapter, we present a program with a slow painting process handled by a server thread. While you probably will find that the program is not especially pleasant to use because of the slow painting, it is very instructive to see how you would farm out painting if you wanted. This program is the most subtle and complex program in the book. When you understand the ins and outs of the synchronization in this program, you can consider yourself multithreading-savvy.

Program prog17-1.c, which is presented in Figure 17.1, displays portions of the Mandelbrot set. Even if you don't know what the Mandelbrot set is, you will almost surely recognize it when you see it displayed. It is one of the darlings of the chaos research community.

Figure 17.1 prog17-1.c

```
prog17-1.c
       This program demonstrates the use of an auxiliary thread to
   paint a window.
       This program displays the Mandelbrot set. We do all of the
   computation as part of the window painting process. As such, a
   paint operation is a time-consuming operation. If we did the
   window painting in the main thread, the presentation manager
   would be effectively hung for the duration of the paint.
   Instead, we use a server thread to do the painting.
#include <stdio.h>
#include <stdlib.h>
#include <string.h>
#include <errno.h>
#include <process.h>
#define INCL_WIN
#define INCL_GPI
#define INCL_DOSPROCESS
#define INCL_DOSSEMAPHORES
#include <os2.h>
#include "mt.h"
/*
* Define a point with floating point coordinates
typedef struct {
    double
    double
                        у;
    } POINTD;
   Define a complex number
typedef struct {
    double
                    r;
    double
                    i;
    } COMPLEX;
   Define the basic PM stuff:
HAB
                        anchorBlock;
HWND
                        mandelbrotWind;
                        frameWind;
HWND
#define MANDELWIND
                        "Mandelbrot Window"
#define MANDELWIND_ID 1000
   Define the basic window parameters
POINTL
                        origin;
```

```
POINTL
                        windowSize;
 ^{\ast} planeLLC and planeSize define the area of complex space to
 * represent.
#define MAX_ITERATIONS 100
POINTD
                        planeLLC = \{-2.0, -1.25\};
POTNTD
                        planeSize = \{2.75, 2.50\};
COMPLEX
                        scale;
^{\prime} * Define the presentation space the paint process will use.
HPS
                        paintPresSpace= NULLHANDLE;
 * Declare the window process for the main window.
FNWP
                        mandelbrotWP;
* Declare the painting server thread
void
                        paintThread(void *);
 * Define the draw box routine
                        drawBox(HPS presSpace, POINTL *llc, POINTL *size);
void
 * Define a semaphore to poke server task
HEV
                        paintSignal;
HMTX
                        paintLock;
HRGN
                        paintRegion[2];
USHORT
                        paintRegionNum;
HEV
                        painterWaiting;
HEV
                        painterRunning;
HEV
                        painterWait;
                        abortPaint= FALSE;
char
void main(void)
{
                        messageQ;
    MMQ
    OMSG
                        message;
    ULONG
                        createFlags;
    USHORT
                        serverTid;
     * Get PM-ready
    anchorBlock= WinInitialize(0);
    messageQ= WinCreateMsgQueue(anchorBlock, 0);
       Create the paint event semaphore. This semaphore is used to
       wake up the paint server thread.
```

Figure 17.1 Continued

```
DosCreateEventSem(
                                       /* Local semaphore, no name */
                    NULL.
                                      /* Addr of semaphore handle */
                    &paintSignal,
                                       /* No options */
                    0,
                                       /* Initially not posted
                    FALSE):
  Create the painter waiting event semaphore. This semaphore is used
  by the WM_BUTTON1CLICK message to synchronize with the painting
* thread.
DosCreateEventSem(
                                     /* Local semaphore, no name */
/* Addr of semaphore handle */
/* No options */
/* Initially not posted */
                    NULL,
                    &painterWaiting,
                    FALSE);
   Create the painter running event semaphore. This semaphore is used
 * by the WM_BUTTON1CLICK message to synchronize with the painting
 * thread.
DosCreateEventSem(
                                      /* Local semaphore, no name */
                                      /* Addr of semaphore handle */
                    &painterRunning,
                                      /* No options
                    0,
                    TRUE):
                                       /* Initially POSTED
  Create the painter wait event semaphore. This semaphore is used
   by the WM_BUTTON1CLICK message to synchronize with the painting
 * thread.
 */
DosCreateEventSem(
                                      /* Local semaphore, no name */
                                      /* Addr of semaphore handle */
                    &painterWait,
                                      /* No options
                                      /* Initially not posted
                    FALSE);
    Create a mutual exclusion semaphore to lock the paint
   communications variables.
DosCreateMutexSem(
                                      /* Local semaphore, no name */
                                      /* Addr of semaphore handle */
                    &paintLock,
                                      /* No options
                                      /* Initially unowned
                    FALSE):
serverTid= _beginthread(
                    paintThread, /* Address of function
                                     /* Don't really give stack
/* Size of stack needed
/* Message to thread
                    NULL,
                    PM STACK SIZE.
                    (PVOID) 0);
    Create our TicTacToe window class
WinRegisterClass(
```

/* Anchor Block

```
MANDELWIND.
                                            /* Class Name
                        mandelbrotWP,
                                            /* Window proc for class*/
                                            /* Redraw on a resize */
                        CS SIZEREDRAW.
                                            /* No user data
    createFlags=
            FCF TITLEBAR
                                FCF SYSMENU
                                                        FCF SIZEBORDER
                            | FCF_SHELLPOSITION |
            FCF MINMAX
                                                        FCF_TASKLIST;
        Create the window
    frameWind= WinCreateStdWindow(
                        HWND DESKTOP.
                                            /* Parent is desk top
                                            /* Make frame visible
                        WS_VISIBLE,
                        &createFlags,
                                            /* Various flags
                                            /* Window Class
                        MANDELWIND,
                        "Prog16-1 - Mandelbrot Set",
                                            /* Title
/* Make client visible
                        WS VISIBLE.
                                            /* Resources in .EXE
                        NULLHANDLE,
                                            /* Window ID
                        0.
                                            /* Client HWND
                        &mandelbrotWind):
     * Bring it up front and activate it.
    WinSetFocus(HWND_DESKTOP, mandelbrotWind);
    while (WinGetMsg(anchorBlock, &message, (HWND) NULL, 0, 0)) {
        WinDispatchMsg(anchorBlock, &message);
        }
        Do the shutdown stuff
    WinDestroyWindow(frameWind);
    WinDestroyMsgQueue(messageQ);
    WinTerminate(anchorBlock);
}
        mandelbrotWP()
        This procedure is the window procedure for a window that
    displays the mandelbrot set. It depends on an auxiliary thread
    to paint the window.
MRESULT EXPENTRY mandelbrotWP(HWND window, ULONG msgNum,
            MPARAM msgParm1, MPARAM msgParm2)
    static HRGN
                        paintMessageRegion;
    static char
                        centerDefined;
    static POINTL
                        center:
    static POINTL
                        boxLLC:
    static POINTL
                        boxSize:
    short
```

anchorBlock,

Figure 17.1 Continued

```
POINTL
                    corner;
POINTL
                    location;
char
                    buff[132];
SIZEL
                    presSpaceSize;
double
                    windowAspect;
double
                    planeAspect;
                    presSpace;
HPS
ULONG
                    postCount;
switch (msgNum) {
case WM_CREATE:
   /*
        Create a presentation space for use by the painting
       thread.
    presSpaceSize.cx= 0;
    presSpaceSize.cy= 0;
    paintPresSpace= GpiCreatePS(anchorBlock, WinOpenWindowDC(window),
                            &presSpaceSize,
                            PU_PELS | GPIF_SHORT | GPIT_NORMAL
                            : GPIA ASSOC);
        Create the two regions we will be using to communicate
     * repaint region information
    for (i = 0; i < 2; i++) {
        paintRegion[i] = GpiCreateRegion(paintPresSpace, 0, NULL);
        Create a region that we will use to get the extent of a
       repaint request.
    paintMessageRegion= GpiCreateRegion(paintPresSpace, 0, NULL);
     * We'll be using #0 first
    paintRegionNum= 0;
    break;
case WM SIZE:
    ^{\prime*} Recalculate the scaling and origin.
    windowSize.x= SHORT1FROMMP(msgParm2);
    windowSize.y= SHORT2FROMMP(msgParm2);
     * Make sure we don't get floating point errors
    if (planeSize.x == 0.0) {
        planeSize.x= .0000001;
    if (planeSize.y == 0.0) {
        planeSize.y= .0000001;
```

261

```
windowAspect = (double) windowSize.x / (double) windowSize.y;
    planeAspect= planeSize.x / planeSize.y;
     * The aspect ratio of the desired portion of the plane might not
       match the aspect ratio of the window. We want to get the
     * largest magnification we can get and still have the whole
     * area shown in the window.
    if (windowAspect >= planeAspect) {
        scale.r= (double) planeSize.y / windowSize.y;
        scale.i= scale.r;
        origin.x= (windowSize.x - planeSize.x / scale.r) / 2;
        origin.y= 0;
    else {
        scale.r= (double) planeSize.x / windowSize.x;
        scale.i= scale.r;
        origin.x= 0;
        origin.y= (windowSize.y - planeSize.y / scale.i) / 2;
    origin.x+= -planeLLC.x / scale.r;
    origin.y+= -planeLLC.y / scale.i;
    break:
case WM MOUSEMOVE:
    * Get a presentation space and display the cursor position.
    * This shows that two threads can be writing to the window
       simultaneously.
    presSpace= WinGetPS(window);
    sprintf(buff, "(%f, %fi)",
                (double) (SHORT1FROMMP(msgParm1) - origin.x) * scale.r,
                (double) (SHORT2FROMMP(msgParm1) - origin.y) * scale.i);
    GpiSetBackMix(presSpace, BM_OVERPAINT);
   location.x= 8;
    location.y= 8;
    GpiCharStringAt(presSpace, &location, strlen(buff), buff);
    /*
* Now see whether we're making a box
    if (centerDefined) {
       * We're outlining a new area in the complex plane.
        * First erase the old box (drawBox uses XOR).
       drawBox(presSpace, &boxLLC, &boxSize);
        * Now figure out the new box dimensions
       corner.x= SHORT1FROMMP(msgParm1);
       corner.y= SHORT2FROMMP(msgParm1);
       if (corner.x < center.x) {
           boxLLC.x= corner.x;
       else {
           boxLLC.x= 2 * center.x - corner.x;
```

Figure 17.1 Continued

```
if (corner.y < center.y) {
            boxLLC.y= corner.y;
        else {
            boxLLC.y= 2 * center.y - corner.y;
        boxSize.x= 2 * abs(corner.x - center.x);
        boxSize.y= 2 * abs(corner.y - center.y);
         * Draw the new box.
        drawBox(presSpace, &boxLLC, &boxSize);
   WinReleasePS(presSpace);
    /*
* Pass message on to default procedure
    return WinDefWindowProc(window, msgNum, msgParm1, msgParm2);
   break:
case WM BUTTON1CLICK:
    * This is either starting or ending a new area of the complex
   if (!centerDefined) {
            We're starting an area. Make it zero size, centered on
         * the cursor.
        center.x= SHORT1FROMMP(msgParm1);
        center.y= SHORT2FROMMP(msgParm1);
        boxLLC.x= center.x;
        boxLLC.y= center.y;
        boxSize.x= 0;
        boxSize.y= 0;
        centerDefined= TRUE;
         ^{'} * Now we want to abort any painting that might be going on.
         * This is rather complex. We want to _know_ that the
         ^{\star} painting thread is waiting on the painterWait semaphore,
         * or just about to wait. To make this happen, we first
         * reset the painterWait and painterWaiting semaphores.
         * This can cause atomicity problems because they are
            only touched by the server thread when abortPaint is true.
            We then set abortPaint. This will abort the painter unless
            it is blocked on paintSignal. We take care of this
            possibility by posting paintSignal. We then wait for
            painterWaiting to be posted, telling us that the painter
         * painterWaiting to be posted, terring as that the is waiting (or about to wait) on painterWait. We then can
         * reset painterSignal (we posted it, but the server
         * might not have been waiting on it).
```

```
* Wait till the painter thread is out of any previous
     * aborting code.
    DosWaitEventSem(
                painterRunning,
                SEM_INDEFINITE_WAIT);
     * Reset painterWaiting so we'll know when it has been
     * signalled.
    DosResetEventSem(
                painterWaiting,
                &postCount);
     ^{\prime} * Reset painterWait so that the paint thread waits on it
     * after it finds abort set.
    DosResetEventSem(
                painterWait.
                &postCount);
     * Set the flag that tells the painter thread to go into
     * abort mode
    abortPaint= TRUE:
     ^{\prime} ^{*} Wake up the paint thread in case it's sleeping
     * so that it goes into abort mode.
    DosPostEventSem(paintSignal);
    ^{/*} ^{*} Wait for the server to get into the abortcheck code
    DosWaitEventSem(
                painterWaiting,
SEM_INDEFINITE_WAIT);
     * Reset paintSignal so that the paint thread waits on it
     * after we release the hold.
    DosResetEventSem(
                paintSignal,
                &postCount);
    * Draw the box
    presSpace= WinGetPS(window);
    drawBox(presSpace, &boxLLC, &boxSize);
    WinReleasePS(presSpace);
else {
    * We're finishing the area definition. Erase the box
    centerDefined= FALSE;
```

presSpace= WinGetPS(window);

Figure 17.1 Continued

```
drawBox(presSpace, &boxLLC, &boxSize);
       WinReleasePS(presSpace);
            Compute the final box
        corner.x= SHORT1FROMMP(msgParm1);
        corner.y= SHORT2FROMMP(msgParm1);
        if (corner.x < center.x) {
            boxLLC.x= corner.x;
            }
        else {
            boxLLC.x= 2 * center.x - corner.x;
        if (corner.y < center.y) {
            boxLLC.y= corner.y;
        else {
            boxLLC.y= 2 * center.y - corner.y;
        boxSize.x= 2 * abs(corner.x - center.x);
        boxSize.y= 2 * abs(corner.y - center.y);
         * Translate it into complex coordinates
        planeLLC.x= (boxLLC.x - origin.x) * scale.r;
        planeLLC.y= (boxLLC.y - origin.y) * scale.i;
        planeSize.x= boxSize.x * scale.r;
       planeSize.y= boxSize.y * scale.r;
           Update the translation parameters by faking
           a resize with no change in size.
        WinSendMsg(window, WM_SIZE,
                        MPFROM2SHORT(windowSize.x, windowSize.y),
                        MPFROM2SHORT(windowSize.x, windowSize.y));
            Create a redraw request
       WinInvalidateRect(window, NULL, FALSE);
        /*
* Wake up the painter
        DosPostEventSem(painterWait);
    break:
case WM_PAINT:
     * This is the central point in the program. We don't want
      to actually paint the window here. It would take too long.
     ^{\star} So, we just add the paint request region to the accumulating
     * region and signal the painting thread.
     * Start by getting the request region
```

```
WinQueryUpdateRegion(
                                  /* Window handle
                window.
                paintMessageRegion);/* Receives paint req.
                                                                */
 * Validate the region, i.e. acknowledge the paint request,
 * so we don't get it over and over and over again. We
 * do this by issuing a WinBeginPaint(). We use this because
 * a) we need a presentation space anyway and b) we want a
 * presentation space with the right clipping for drawing
    the selection box.
    We need a presentation space for the combine region call.
   We cannot use paintPresSpace because it might be busy.
presSpace= WinBeginPaint(window, NULLHANDLE, NULL);
 * First erase the area.
GpiSetBackColor(presSpace, CLR BLACK);
GpiSetBackMix(presSpace, BM_OVERPAINT);
GpiPaintRegion(presSpace, paintMessageRegion);
   Draw the box, if there is one
if (centerDefined) {
    drawBox(presSpace, &boxLLC, &boxSize);
   Now lock the communications variables
DosRequestMutexSem(
                                    /* Semaphore handle
                paintLock,
                SEM_INDEFINITE_WAIT);/* Don't time out
 * Add the paint request to the current outstanding paint
   region.
if (RGN_ERROR == GpiCombineRegion(presSpace,
                paintRegion[paintRegionNum],
                paintRegion[paintRegionNum],
                paintMessageRegion,
                CRGN OR)) {
    ULONG error;
    error= WinGetLastError(anchorBlock);
WinEndPaint(presSpace);
 * Now post the semaphore. This will signal the paint thread
 * that there's something there.
DosPostEventSem(paintSignal);
 * And finally, release the lock
DosReleaseMutexSem(paintLock);
                                     /* Semaphore handle
break:
```

Figure 17.1 Continued

```
case WM_ERASEBACKGROUND:
        return (MRESULT) FALSE;
        break;
    default:
        return WinDefWindowProc(window, msgNum,
                    msgParm1, msgParm2);
        }
    return FALSE;
}
        paintThread()
        This is the thread that actually does the painting of the
   Mandelbrot window.
void
        paintThread(void * _)
{
    long
                        colorTable[]= {
            CLR_BLUE,
            CLR_DARKBLUE,
            CLR_PINK,
            CLR_DARKPINK,
            CLR_RED,
            CLR_DARKRED,
            CLR_YELLOW,
            CLR_BROWN,
            CLR_GREEN,
            CLR_DARKGREEN,
            CLR_DARKCYAN,
            CLR_CYAN
            };
    POINTL
                        location;
    static HRGN
                        clipRegion= NULLHANDLE;
    long
                        í;
    long
                        j;
    ULONG
                        postCount;
    ULONG
                        color;
    COMPLEX
                        С;
    COMPLEX
                        Ζ;
    COMPLEX
                        z2;
    USHORT
                        iterations;
    HRGN
                        oldRegion;
    while (TRUE) {
            See if we're aborting
        if (abortPaint) {
                Make the main thread pend if he tries to
                abort us a second time.
```

```
DosResetEventSem(
                 painterRunning,
                 &postCount):
    /*
* Reset the abort bit
    abortPaint= FALSE;
     * First signal that we're about to start waiting
    DosPostEventSem(painterWaiting);
    /*

* Now wait till we're synched up
    DosWaitEventSem(
                                      /* Wait till we're clear
                 painterWait,
                 SEM_INDEFINITE_WAIT);
     * Tell the main thread we're out of this code path.
    DosPostEventSem(painterRunning);
    Wait for a paint request.
DosWaitEventSem(
                                     /* Signalling semaphore
                 paintSignal,
                 SEM_INDEFINITE_WAIT);/* Forever
 * OK, we need to lock up the communications variables
* _before_ we reset the event semaphore. Otherwise, the 
* semaphore might get signalled after we reset it but before
* we've switched paintRegionNum. This could cause us to
 * pick up a bogus region.
DosRequestMutexSem(
                 paintLock.
                                      /* Semaphore handle
                 SEM_INDEFINITE_WAIT);/* Don't time out
DosResetEventSem(
                 paintSignal,
                                     /* We ignore this
                                                                    */
                 &postCount):
 * Make a copy of the clip region so we can check points
 * for inclusion. Once we have assigned the original as the
^{\star} clipping region, we will be unable to access it any more.
if (clipRegion == NULLHANDLE) {
    clipRegion= GpiCreateRegion(
                paintPresSpace,
                 NULL):
else {
```

Figure 17.1 Continued

```
GpiSetRegion(
                paintPresSpace,
                clipRegion,
                0.
                NULL):
GpiCombineRegion(paintPresSpace,
                clipRegion,
                clipRegion.
                paintRegion[paintRegionNum],
                CRGN OR):
   Now set our presentation space to clip based on
   the paint region.
GpiSetClipRegion(
                paintPresSpace,
                paintRegion[paintRegionNum],
                &oldRegion);
  OK, now swap regions, paintRegionNum is the region that
   the window procedure has been augmenting with each paint.
* We now will clear out the other region and switch
* paintRegionNum. This will allow us to use the region that
* the window procedure has been updating and give the
   window procedure an empty region to start updating.
paintRegionNum^= 1;
GpiSetRegion(
                paintPresSpace,
                paintRegion[paintRegionNum],
                Ö.
                NULL);
   We now can release the lock. The other thread will not
   touch our region.
DosReleaseMutexSem(paintLock);
   OK, let's get down to some painting. Iterate over all
   of the pixels in the window.
for (j= 0; !abortPaint && j < windowSize.y; j++) {
    for (i= 0; !abortPaint && i < windowSize.x; i++) {
        location.x= i;
        location.y= j;
         * Use our local copy of the paint region to determine
            whether the point is even to be repainted. If not,
         * don't do all of the computation.
        if (GpiPtInRegion(paintPresSpace, clipRegion, &location)
                    == PRGN_OUTSIDE) {
            continue;
```

```
}
    OK, we've got to do some computation. The formula we pre
    iterating on is
        Z = Z_2 + C
    where z and c are complex numbers. If this doesn't
    blow up (approach infinity), c is in the mandelbrot set.
 * It turns out that if |z| > 2 at any point, then it will
    blow up. We count the number of iterations it takes for
   !z! to become > 2. We paint based on this number.
 * First translate the pixel into a point in the
 * complex plane.
c.r= (double) (i - origin.x) * scale.r;
c.i= (double) (j - origin.y) * scale.i;
   Now iterate
z.r = 0.0;
z.i = 0.0:
for (iterations= 1;
        iterations < MAX ITERATIONS:
            iterations++) {
    z2.r= z.r * z.r - z.i * z.i;
    z2.i= z.r * z.i * 2.0;
    z.r = c.r + z2.r;
    z.i = c.i + z2.i;
    if (z.r*z.r + z.i*z.i > 4.0) {
        break;
    }
    Choose a color
if (iterations >= MAX_ITERATIONS) {
    color= CLR BLACK;
    }
else {
    * Do a pseudo-log kinda-thingy
    color= 0;
    while (iterations > 7) {
        color+= 4;
        iterations>>= 1;
    color+= iterations & 0x3;
    color= colorTable[color % 12];
    Set the desired color...
GpiSetColor(paintPresSpace, color);
    ... and paint the pixel
```

Figure 17.1 Continued

```
GpiSetPel(paintPresSpace, &location);
        }
}
        drawBox()
        This procedure XORs a box of the specified size at the
    specified location onto the specified presentation space.
    It is used for zooming in on areas of the mandelbrot map
void drawBox(HPS presSpace, POINTL *llc, POINTL *size)
{
    POINTL
                        location;
    GpiSetMix(presSpace, FM_XOR);
    GpiSetColor(presSpace, 8);
    location.x= llc->x;
    location.y= llc->y;
    GpiMove(presSpace, &location);
    location.y+= size->y;
    GpiLine(presSpace, &location);
    location.x+= size->x;
    GpiLine(presSpace, &location);
    location.y-= size->y;
    GpiLine(presSpace, &location);
    location.x-= size->x;
    GpiLine(presSpace, &location);
```

Just so you know about as much about it as we do, we'll give a little background. A complex number is a number having two components: a real part and an imaginary part. The real part is a normal number like we are used to. The imaginary part is a real number multiplied by a special number called i. This number, i, is defined to be the square root of -1. We know that no real number can be the square root of -1. Hence the term "imaginary." The following is an example of how a complex number is written: 5.4 + 3.4i. We can multiply complex numbers as we would multiply real numbers. The product of a + bi and c + di is $(a \times c) + (a \times di) + (c \times bi) + (bi \times di)$. Note that $bi \times di$ is $b \times d \times i \times i$. Because i is the square root of -1, it's $-b \times d$. (If you're experiencing math anxiety, feel free to skip ahead, we won't be insulted.)

To compute the Mandelbrot set, we begin with a complex number c. Starting with z equal to 0, we iterate, setting z to $z^2 + c$. For some values of c, z will get ever fur-

ther from the origin 0 + 0i, approaching infinity. For other values of c, z will never get above a certain distance from the origin. These latter points form the Mandelbrot set. It turns out that if |z|, which is the distance of z from the origin (for z = a + bi, |z| is the square root of $a^2 + b^2$), ever gets larger than 2, it eventually will "blow up."

The complex numbers can be represented in a plane with one coordinate being the real component and the other component being the imaginary component. Our program displays a portion of the complex plane. It assigns a color to each point. Points in the Mandelbrot set are displayed black. Other points are displayed in various colors depending on how quickly they blow up.

Enough mathematics, let's talk multithreading!

The program consists of two threads: the main thread, which is a message queue thread and owns program window, and a server thread, which is used to compute the Mandelbrot set and paint the points. The outline of the program is this: when a paint message is sent to the window, the message is handled by the window procedure. Rather than doing the painting itself, which could take an enormous amount of time, it sends the paint request to the server thread. The server thread then paints the window in a leisurely fashion. One complicating factor is that we want to be able to abort the repainting that the server thread is doing. The program provides the user with the capability of zooming in on areas of the complex plane. This is done by clicking button 1 to define the center of the new area of interest and clicking button 1 a second time to define any corner of the area. When button 1 is clicked, we abort the current painting operation (if there is one). We do this for two reasons. First, the painting will prove useless, because we are zooming in on a new area. Second, the painting will interfere with the frame box that we draw as we track the mouse.

Initially, let's ignore the need for aborting a repaint. To pass paint requests from the window procedure to the server thread, some synchronization is required. A first guess would be to follow our producer-consumer model with the main thread producing paint requests and the server thread consuming them.

There are two drawbacks to this approach. The first is that we generally need to limit the number of unconsumed resources that we produce. This would mean that we could have only a finite number of paint requests outstanding. After that, the window procedure would need to wait for resources to generate a paint request. This waiting is what we're trying to avoid in the first place.

The second problem is that we don't want to paint each and every paint request. In the time since we last serviced a paint request, many paint requests might have been generated and they might overlap. We would like to avoid, where possible, painting the same areas over and over. In a producer-consumer model, we might solve this problem by having the server gather all of the paint requests available to it and consolidate them into one request. This idea is the seed for a much better technique that we use in prog17-1.

A paint request is a request to repaint a region. A region is an abstract data type supported by the Presentation Manager. A region can be thought of as a collection of rectangles that define a subset of a window. Regions can be created by supplying a set of rectangles, and a set of rectangles can be obtained from a region. More importantly, regions can be combined using set intersection or set union. If we union two regions together, we get one region that contains all of the points in the two regions.

This is how we consolidate paint requests: we union them together to produce one paint request that paints all of the areas that need to be repainted.

Performing this consolidation on the server (consumer) end would not solve the problem of having large numbers of paint requests outstanding at any given time. Instead, we want to do the consolidation in the window process (the producer). In previous producer-consumer programs, the producer has produced an object, then sent it to the consumer without touching it again. We've been able to get away with few mutex semaphores for this reason. Here, we want to make a paint request available to the consumer; however, if another request comes along before the first is consumed, we want to combine the two. To protect the state, we will need a mutex semaphore.

Here's how it works: we use an array of two regions, paintRegion[]. This is analogous to double-buffering. At any given time, one region is in use by the server and one is in use by the window process. An index, paintRegionNum, indicates which region is in use by the window procedure. On a paint message, the window procedure performs the following steps:

The process is fairly simple. We get the new region, add it to the old region, and signal the painter that there is a paint request outstanding. This process is protected by a lock to make sure that the threads don't step on each other's regions.

The algorithm on the server side is a little more complex, because it is the server that switches the region ownership:

```
loop
wait on paintSignal
lock paintLock
reset paintSignal
copy paintRegion[paintRegionNum] to clipRegion
set clipping region for presentation space
to paintRegion[paintRegionNum]
toggle paintRegionNum
set paintRegion[paintRegionNum] to empty
unlock paintLock
paint
endloop
```

Note that the first thing we do after waking up from the paintSignal wait is to lock paintLock. We do this even before resetting paintSignal. We want to do things in this order because, otherwise, the window procedure might add a new request to paintRegion[paintRegionNum] and signal paintSignal after we reset the semaphore but before we toggle paintRegionNum. This would cause us to service the new paint request on this round. This is no problem, but the semaphore would be signalled, and we would go around a second time with an empty region. We copy the region to a local region, clipRegion. When a region is set as a clip region of a presentation space, it cannot be used for anything else. We will want to check

whether a point is in the current clip region when painting. This will free us from doing a lot of complex number multiplication for points that aren't even in the paint region. After making a copy of the paint region, we make it the clip region for the painting presentation space. We then are ready to toggle paintRegionNum. This will cause the window procedure to begin accumulating paint requests in the other region. Before we can allow this to happen, though, we must empty out the other region. The painting process itself is quite simple. Because, in this instance, we are painting pixel-by-pixel, we check each pixel before we compute it to see whether to bother painting it or not.

This is a reasonable and reliable method of painting in an auxiliary thread. Let's now look at the other problem we have in our program: aborting a paint.

This program gives the user the ability to select an area of the window to zoom in on. When the user has selected an area, the whole window becomes invalid and we would like to be able to abort any painting that might be going on. We'd like to abort painting before we start drawing the frame box. We use XOR for drawing and undrawing the frame box. Painting at the same time, while possible, would cause residue to be left over from the frame box. Aborting the painting requires some rather fancy footwork. First of all, note that it is not enough to have the painting abort at some future time. The window procedure needs to know when the painting has aborted before it can start drawing the frame box.

We solved this problem in chapter 12, and we use the same solution here. The basic idea of the synchronization is this: a variable abortPaint is used to tell the server thread to abort the current paint operation. When it finds this variable set, it stops any painting, signals painterWaiting, telling the main thread that it's about to block, and waits on a painterWait.

To abort the paint, the window procedure first waits for painterRunning to be posted. This ensures that the server is not in the middle of a previous abort. It then resets painterWait and painterWaiting, sets abortPaint, and signals paintSignal. It then waits for painterWaiting to be signalled. It then knows that the painter is waiting on the painterWait semaphore or is about to. When painterWait is posted, the paint thread ends up in it's normal state. The new algorithm for the painter looks like this:

```
* if (abortPaint)
      reset painterRunning
       signal painterWaiting
       wait on painterWait
       post painterRunning
  wait on paintSignal
  lock paintLock
  reset paintSignal
  copy paintRegion[paintRegionNum] to clipRegion
  set clipping region for presentation space
       to paintRegion[paintRegionNum]
  toggle paintRegionNum
  set paintRegion[paintRegionNum] to empty
  unlock paintLock
* paint (break out if abortPaint is set)
endloop
```

Only the lines marked with an asterisk are new. The modification is simple enough. When it starts a frame box, the window process aborts a paint with the following code:

wait on painterRunning reset painterWait reset painterWaiting set abortPaint post paintSignal wait on painterWaiting reset paintSignal

The resetting of painterWait and painterWaiting are done first because the server thread touches these semaphores only when abortPaint is set. We then set abortPaint. This will abort the painting unless the server thread is blocked on paintSignal. We post paintSignal to take care of this possibility. We then wait for the server to post paintWaiting, telling us that it is blocked or is about to be blocked on painterWait. We then reset paintSignal in case the server was not waiting on it when we posted it. We do not want an extraneous post.

When the user has finished selecting an area, the painter is reenabled by simply posting to painterWait. This wakes up the server thread. It then will go into the wait for paintSignal, which will be signalled if there has been a paint request. In this program, there always is because we've invalidated the window just before signalling painterWait.

We encourage you to study this program carefully. There are many rather subtle techniques being used here.

Summary

One of the messages that a window procedure must respond to in a timely manner is a WM_PAINT message. If painting the window is a lengthy process, it must be handled by a server thread. In this chapter, we have presented a reasonably efficient solution to passing paint messages to a server thread without doing an inordinate amount of redundant painting.

Chapter

18

Using Multiple Message Queue Threads

In the previous two chapters, we have used only one message queue thread in our programs. It generally is not necessary to have more than one thread for processing messages. Because message processing should be fairly rapid, there isn't much need for overlapping message processing using multiple message queue threads. However, for completeness, we present an application here which has four threads, each with its own window, each with its own message queue.

Program prog18-1.c, which is presented in Figure 18.1, really is a toy program, but it demonstrates some useful techniques. The program has four windows that it tiles on the screen. There are 15 circles of different colors that are passed from window to window. The colored circles are displayed in the windows. At any given time, there will be one circle of each color on the screen. What makes this program interesting is that the four windows are owned by four different threads. In essence, each thread is its own Presentation Manager applications, although they all share an address space.

The main thread creates four auxiliary threads. Each thread becomes a message queue thread by creating a message queue. Each thread then creates a window and goes into a message loop. The window processing is quite simple. Each window owns some subset of the 15 colored circles. A paint request paints the circles that window owns. Each window has a timer set to trigger every second. On each timer tick, each window gives one of its circles (if it has any) to another window. The circle and the window are chosen at random. What you see on the screen is four windows with 15 colored circles distributed between them. Every second each window passes a circle to another window.

There is one window procedure that is shared by all four threads for all four windows. Presentation Manager programmers often make use of static variables in window procedures to maintain state from one message to the next. An alternative

approach, used particularly when there are several windows being handled by one window procedure, is to attach user data to each window's WinSetWindowPtr() and accessing it via WinQueryWindowPtr(). In this program, however, we extend the idea of static data to multithreading. We use the _threadstore() function to maintain access to a per-thread data area. Much of the data that we put into the per-thread area is data that was global or static in other PM programs (for example, the anchor block and frame and client window handles).

We use semaphores to communicate between the main thread and the PM threads. It is necessary for the main thread to wait until the other threads are ready before distributing the circles to the windows. Each thread has a semaphore that it posts to signal its readiness. The main thread waits on each semaphore. When it's done waiting, all of the threads are ready.

Figure 18.1 prog18-1.c

```
prog18-1.c
        This program demonstrates using multiple message queue
    threads to handle multiple windows.
        Four threads are created. Each thread has its own message
    queue and its own window. Fifteen colored circles are
    distributed amongst the threads. Each second they pick one of
    their circles at random and pass it on to another thread.
#include <process.h>
#include <stdlib.h>
#include <stdio.h>
#define INCL_WIN
#define INCL_GPI
#define INCL_DOSPROCESS
#define INCL_DOSSEMAPHORES
#include <os2.h>
#include "mt.h"
 * Define the messages we'll be using
#define WM_TAKE_CIRCLE
                            WM_USER+1
\#define\ WM\_START\_GOING
                            WM_USER+2
 * Define per-thread structure
typedef struct {
    USHORT
                    threadNum:
    HAB
                    anchorBlock;
                    frameWind:
    HWND
    HWND
                    clientWind;
```

```
POINTL
                    windowSize;
                    haveCircle[16];
    char
       PER THREAD AREA;
    Define a handy macro for accessing the per-thread area
#define PTA
                    (*(PER THREAD AREA **) threadstore())
    Declare the semaphores that will tell the main thread
    that the others are ready.
*/
HEV
                    threadReady[4];
   Create an array of the window IDs
HWND
                    clientWind[4];
/
* Prototype the procedures
void drawCircle(HPS presSpace, USHORT color);
void eraseCircle(HPS presSpace, USHORT color);
FNWP clientWP;
void windowThread(void *threadNumV);
void main()
    USHORT
                        i;
    USHORT
                        target;
    ULONG
                        tid[4];
       First create the window threads
    for (i = 0; i < 4; i++) {
        DosCreateEventSem(
                                             /* Unnamed semaphore
                                             /* Handle
                        &threadReady[i],
                        0,
                                            /* No options not shared*/
                                            /* Initially reset
                        FALSE):
        tid[i]= _beginthread(
                                             /* Start Address
                        windowThread,
                                             /* Dummy stack addr
                        NULL,
                        PM_STACK_SIZE,
                                             /* Stack size
                        (void *) i);
                                             /* Thread number
       Wait till they're all ready
    for (i = 0; i < 4; i++) {
       DosWaitEventSem(
                        threadReady[i],
                        SEM_INDEFINITE_WAIT);
        }
```

Figure 18.1 Continued

```
^{\prime*} ^{} OK, POST the various colors to the windows.
   for (i = 1; i < 16; i++) {
        * Pick a window at random
        target= rand() % 4;
        WinPostMsg(clientWind[target], WM_TAKE_CIRCLE,
                        MPFROMSHORT(i),
                        MPFROMSHORT(0));
        }
       Now wait for them all to terminate
   for (i = 0; i < 4; i++) {
        DosWaitThread(&tid[i], DCWW_WAIT);
}
        windowThread()
        This is the main routine of the four auxiliary threads. It
    creates an anchor block, a message queue, and a window, then
    goes into a message loop. Each thread is essentially an
    independent PM application.
void windowThread(void *threadNumV)
                        messageQ;
   OMSG
                        message;
   ULONG
                        createFlags;
    char
                        className[24];
    char
                        title[24];
    SWP
                        windPos;
    * Allocate the per-thread structure.
*/
    PTA= malloc(sizeof(PER_THREAD_AREA));
       Set the thread number
    PTA->threadNum= (USHORT) threadNumV;
    /*
* Get PM-ready
    PTA->anchorBlock= WinInitialize(0);
    messageQ= WinCreateMsgQueue(PTA->anchorBlock, 0);
       Create our window class
```

```
*/
sprintf(className, "prog18-1.%d", PTA->threadNum);
sprintf(title, "prog18-1 - Window #%d", PTA->threadNum+1);
WinRegisterClass(
                    PTA->anchorBlock, /* Anchor Block
                                        /* Class Name
                    className,
                                       /* Window proc for class*/
                    clientWP,
                                       /* Redraw on a resize
                    CS_SIZEREDRAW,
                                        /* No user data
    Create the window
createFlags=
        FCF_TITLEBAR
                            FCF_SYSMENU
                                              + FCF SIZEBORDER
       FCF MINMAX
                           FCF_TASKLIST;
PTA->frameWind= WinCreateStdWindow(
                                        /* Parent is desk top
                    HWND DESKTOP,
                                        /* Make frame invisible */
                    &createFlags,
                                       /* Various flags
                                       /* Window Class
                    className,
                                       /* Title
                    title,
                                        /* Make client invisible*/
                    WS_VISIBLE,
                                        /* Resources in .EXE
                    NULLHANDLE,
                                        /* Window ID
                    &PTA->clientWind); /* Client HWND
    Set the position.
    First get the size of the screen.
WinQueryWindowPos(HWND_DESKTOP, &windPos);
   Now tile the desktop
WinSetWindowPos(PTA->frameWind, HWND_TOP, (PTA->threadNum & 1) ? windPos.cx/2 : 0,
                    (PTA->threadNum & 2) ? 0 : windPos.cy/2,
                    windPos.cx/2,
                    windPos.cy/2,
                    SWP_MOVE | SWP_SIZE | SWP_SHOW);
   Tell the main thread we're ready
clientWind[PTA->threadNum] = PTA->clientWind;
DosPostEventSem(threadReady[PTA->threadNum]);
   Get the window going
WinSendMsg(PTA->clientWind, WM_START_GOING,
           MPFROMSHORT(0), MPFROMSHORT(0));
    Go into the message loop
while (WinGetMsg(PTA->anchorBlock, &message, (HWND) NULL, 0, 0)) {
    WinDispatchMsg(PTA->anchorBlock, &message);
```

```
Figure 18.1 Continued
```

```
Shut down
    WinDestroyWindow(PTA->frameWind);
    WinDestroyMsgQueue(messageQ);
    WinTerminate(PTA->anchorBlock);
}
        clientWP()
        This is the window procedure for each of the windows.
MRESULT EXPENTRY clientWP(HWND window, ULONG msgNum,
            MPARAM msgParm1, MPARAM msgParm2)
{
    USHORT
                        color;
    USHORT
                        target;
                        presSpace;
    USHORT
                        myColors[16];
    USHORT
                        numMyColors;
    switch (msgNum) {
    case WM_CREATE:
        /*

* Set us up to own no colors
        for (color= 0; color < 16; color++) {
            PTA->haveCircle[color]= FALSE;
        break;
    case WM_START_GOING:
        /*

* Start a window timer
        WinStartTimer(
                        PTA->anchorBlock,
                        PTA->clientWind.
                                            /* Use for timer ID
                        PTA->threadNum,
                                             /* Every 1 second
                        1000):
        break;
    case WM_SIZE:
         * Record the window size
        PTA->windowSize.x= SHORT1FROMMP(msgParm2);
        PTA->windowSize.y= SHORT2FROMMP(msgParm2);
        break;
    case WM_PAINT:
```

```
Start painting
    presSpace= WinBeginPaint(window, NULLHANDLE, NULL);
     * For each color, if we have it, draw it
    for (color= 0; color < 16; color++) {
        if (PTA->haveCircle[color]) {
            drawCircle(presSpace, color);
        }
   /*
* Free up the presentation space
    WinEndPaint(presSpace);
    break;
case WM_TAKE_CIRCLE:
    * Someone's passed a circle to us
     * Note that we have the cicle now
    color= (USHORT) msgParm1;
    PTA->haveCircle[color]= TRUE;
     * Get a presentation space and draw the circle
    presSpace= WinGetPS(window);
    drawCircle(presSpace, color);
    WinReleasePS(presSpace);
    break:
case WM_TIMER:
    * We want to give away a circle.
    ^{\star} First come up with a list of colors we have
    numMyColors= 0;
    for (color= 0; color < 16; color++) {
       if (PTA->haveCircle[color]) {
           myColors[numMyColors++]= color;
       }
    * If we don't have any circles to give away, don't
    if (numMyColors == 0) {
       break;
       }
       Choose one of our circles at random, and give it
    * to a random window.
   color= myColors[rand() % numMyColors];
```

Figure 18.1 Continued

```
do {
            target= rand() % 4;
           } while (target == PTA->threadNum);
           Note that we do not have that circle any more
       PTA->haveCircle[color]= FALSE;
        * Take it off the screen
        presSpace= WinGetPS(window);
        eraseCircle(presSpace, color);
       WinReleasePS(presSpace);
           Pass it to another thread
        WinPostMsg(clientWind[target], WM_TAKE_CIRCLE,
                   MPFROMSHORT(color),
                   MPFROMSHORT(0));
        break;
    case WM_ERASEBACKGROUND:
        return (MRESULT) TRUE;
        break;
    default:
        return WinDefWindowProc(window, msgNum,
                   msgParm1, msgParm2);
    return FALSE;
        drawCircle()
        This procedure draws a circle in the appropriate place.
   Each color has a unique position. The colors go across four
    to a row, four rows:
          1
              2
                   3
          5 6
                  7
       8
          9 10 11
       12 13 14 15
void drawCircle(HPS presSpace, USHORT color)
   USHORT
                       row:
   USHORT
                       col;
    POINTL
                       location;
    * Figure out the row and column from the color
```

```
row= color % 4;
    col= color / 4:
       Figure out the location in the window from the row
       and column.
    location.x= col * PTA->windowSize.x / 4 + PTA->windowSize.x / 8;
    location.y= row * PTA->windowSize.y / 4 + PTA->windowSize.y / 8;
       Draw the circle of the appropriate color
    GpiMove(presSpace, &location);
    GpiSetColor(presSpace, color);
    GpiFullArc(presSpace, DRO OUTLINEFILL,
            MAKEFIXED(min(PTA->windowSize.x, PTA->windowSize.y) / 10, 0));
}
        eraseCircle
void eraseCircle(HPS presSpace, USHORT color)
    USHORT
                        row;
    USHORT
                        col:
    POINTL
                        location:
    row= color % 4;
    col= color / 4;
    location.x= col * PTA->windowSize.x / 4 + PTA->windowSize.x / 8;
    location.y= row * PTA->windowSize.y / 4 + PTA->windowSize.y / 8;
    GpiMove(presSpace, &location);
    GpiSetColor(presSpace, CLR_BACKGROUND);
    GpiFullArc(presSpace, DRO_OUTLINEFILL,
            MAKEFIXED(min(PTA->windowSize.x, PTA->windowSize.y) / 10, 0));
}
```

To communicate between the PM threads, specifically to pass circles from one to the next, we do not use semaphores. The value of semaphores is that they allow a thread to block until an event occurs. We do not want message processing to block. You will note that the only blocking semaphore calls that we have used in the message threads of PM applications have been mutex semaphore requests. We can do this because the mutex semaphores were not held for lengthy periods of time. They were used to protect short code segments. Waiting for events to occur is another thing. This can take a potentially long time.

Instead of semaphores, we use messages. We define a message wM_TAKE_CIRCLE to pass a circle to another window. Note that, although we could send the message from thread to thread, we post it instead. This prevents a possible deadlock. Suppose that all of the threads are in their wM_TIMER processing and that thread 0 sends a circle to thread 1, thread 1 to thread 2, thread 2 to thread 3, and thread 3 back to thread

284 Multithreading and the Presentation Manager

0 all at the same time. If these messages are posted, there is no problem. They are added to the appropriate queues, and the WM_TIMER message can complete. Sending, however, waits for the sent message to complete. Because all of the threads are processing WM_TIMER messages, they are not prepared to process WM_TAKE_CIRCLE messages. Thread 0 will wait for thread 1, which will wait for thread 2, which will wait for thread 3, which will wait for thread 0. In short, nothing will happen.

Summary

A single process can have multiple message queue threads. Each message queue thread functions much like a separate application from the point of view of the Presentation Manager. However, the threads share an address space and can thus communicate more easily. We have presented a simple example of using multiple message queue threads and passing messages between them.

Epilogue

Please feel free to lift code segments from the programs in this book for use in your own programs. While your applications probably will require unique multithreading techniques not presented in the book, there might be mechanisms illustrated that will serve as good starting points for you.

We're always interested in hearing how readers feel about the book and the programs in it. Feel free to write us via the publisher if you have any comments or code you'd like to share with us. We'll do our best to reply as time permits.

Namasté, Len and Marc

Index

_beginthread(), 33, 41, 42 _endthread(), 41, 43 _threadstore(), 33, 39, 44, 276 A abortPaint, 273, 274 address space, 3 atomic operation, 10 B buffers, 181-182, 188, 193-194, 211 busy waiting, 17 C central processing unit (see CPU) complex numbers, 270-274 CPU, 3 CPU state, 4 critical sections, 10-11 D	DosReleaseMutexSem(), 71, 72, 75, 89 DosRequestMutexSem(), 71, 72, 75, 78, 81, 90 DosResetEventSem(), 91, 92, 103 DosResumeThread(), 55, 62 DosSetPriority(), 64, 69-70 DosSleep(), 106, 145, 158 DosStartTimer(), 151 DosStopTimer(), 152 DosSuspendThread(), 55, 61 DosWaitEventSem(), 91, 92, 104, 158 DosWaitMuxWaitSem(), 125-126 DosWaitThread(), 48 DosWrite(), 35 DosWriteQueue(), 143 double-buffering, 181 E emptyBuffAvail, 188 event queues, 213 event semaphores, 19-20, 91-104 event-driven input, 213-222
deadlocks, 26 DosAddMuxWaitSem(), 118 DosAsyncTimer(), 150 DosCloseEventSem(), 98 DosCloseMutexSem(), 84 DosCloseMutexSem(), 119 DosCloseQueue(), 134 DosCreateEventSem(), 99 DosCreateMuxWaitSem(), 120-121 DosCreateMuxWaitSem(), 120-121 DosCreateMuxWaitSem(), 125 DosCreateMuxWaitSem(), 125 DosCreateThread(), 33, 35, 41, 45-46 DosDeleteMuxWaitSem(), 122 DosEnterCritSec(), 11, 49, 52, 162, 166, 176 DosExitCritSec(), 11, 49, 53, 162, 166, 176 DosKillThread(), 35, 41, 47 DosOpenEventSem(), 91, 100 DosOpenMuxWaitSem(), 100 DosOpenMuxWaitSem(), 106, 123 DosOpenMuxWaitSem(), 106, 123 DosOpenQueue(), 136 DosPeekQueue(), 137-138 DosPostEventSem(), 91, 94, 101 DosPurgeQueue(), 139 DosQueryMutexSem(), 88 DosQueryMuxWaitSem(), 102 DosQueryMuxWaitSem(), 124 DosQueryQueue(), 142 DosReadQueue(), 140-141	F FIFO queues, 127 fullBuffAvail, 188 I input/output (I/O), overlapped, 5 instruction pointer (IP), 4 L LIFO queues, 127 M Mandelbrot set, 270-274 message queues, 22-23, 275-284 mouse systems, 213 multiple wait semaphores, 105-126 multitasking, vs. single, 3-5 multithreading, 3-6 complex example, 201-212 event-driven input, 213-222 producer/consumer problem, 181-199 writing simple programs, 155-180 mutual exclusion semaphores, 13, 71-90 N numbers, complex, 270-274 numBuffersFull, 193

overlapped I/O, 5 P ROG3-3. C, 37-39 PROG3-4. C, 39-41 PROG4-1. C, 50-51 PROG5-1. C, 56-57 PROG6-1. C, 64-67 PROG6-2. C, 67-68 PROG6-1. C, 64-67 PROG6-2. C, 67-68 PROG7-2. C, 75-78 PROG7-2. C, 75-78 PROG7-2. C, 75-78 PROG7-3. C, 78-81 PROG7-2. C, 75-78 PROG7-2. C, 19-10 PROG1-2. C, 130-13 PROG1-3. C, 18-10 PROG1-2. C, 130-13 PROG1-3. C, 18-10 PROG1-2. C, 130-13 PROG1-3. C, 18-10 PROG1-2. C, 130-13 PROG1-1. C, 126-15 PROG1-2. C, 130-13 PROG
PainterWait, 273, 274 painterWaiting, 273, 274 painterWaiting, 273, 274 painterWaiting, 273, 274 paintlerWaiting, 273, 274 paintleok, 272 paintRegion, 272 paintRegionNum, 272 paintRegion Num, 272 paintRegionNum, 272 paintRegion Num, 272 paintRegion 272 paintRegion Num, 272 paintRegion Num, 272 paintRegion Num, 272 paintRegion Num, 272 paintRegion 273 paint(10, 61-60 paintregion 273 paintregion 272 paintRegion 273 paintregion 274 paintregion 275 paintregion 275
painterWait, 273, 274 painterWaiting, 273, 274 painterWaiting, 273, 274 painting, 255-274 paintlack, 272 paintRegion, 272 paintRegionNum, 272 paintRegionNum, 272 paintSignal, 272, 273, 274 preemptive scheduler, 4-5 Presentation Manager message queues, 275-284 multithreaded painting, 255-274 using server threads under, 225-253 priority levels, 63-70 priority queues, 127 processes, 3-4 lightweight, 5 processEvent(), 222 program counter, 4 programs and programming complex example of, 201-212 definition, 3 event-driven input, 213-222 producer/consumer problem, 181-199 with threads, 7-29 writing simple multithread, 155-180 Q queues, 127-143 event, 213 FIFO, 127 message, 22-23, 275-284 priority, 127 types of, 127 R race condition, 10 repeated-interval timer, 145 ringBufferEmptx, 188 ringBufferEmptx, 188 ringBufferEmptx, 188 ringBufferEmptx, 188 ringBufferEmptx, 188 ringBufferEmptx, 188 scheduling, 4 two-level, 6 semaphores event, 19-20, 91-104 multiple wait 105-196 PROG5-2.C, 57-59 PROG6-2.C, 67-68 PROG6-1.C, 64-67 PROG6-2.C, 67-68 PROG7-3.C, 78-78 PROG7-3.C, 78-78 PROG7-3.C, 78-81 PROG7-3.C, 78-78 PROG7-3.C, 78-78 PROG7-3.C, 78-81 PROG7-3.C, 78-8 PROG7-3.C, 78-81 PROG9-1.C, 106-10 PROG9-2.C, 109-113 PROG9-1.C, 106-10 PROG9-2.C, 1
painterWaiting, 273, 274 paintling, 255-274 paintling, 255-274 paintling, 255-274 paintling, 272 paintRegion, 272 paintRegion, 272 paintRegionNum, 272 paintRegion, 274 paintRegion, 274 paintRegion, 275 paintRegion, 274 paintRegion, 274 paintRegion, 275 paintRegion, 275 paintRegion, 275 paintRegion, 274 paintRegion, 275 paintRegion, 285 paintRegion, 275 paintRegion, 285 paintRegion, 285 paint
painting, 255-274 paintLock, 272 paintRegion, 272 paintRegion, 272 paintRegionNum, 272 page 2, 2, 2, 2, 2, 2, 2, 2, 2, 2, 2, 2, 2,
paintLock, 272 paintRegion, 272 paintRegion, 272 paintRegionNum, 272 paintRiginal, 272, 273, 274 preemptive scheduler, 4-5 presentation Manager message queues, 275-284 multithreaded painting, 255-274 using server threads under, 225-253 printf(), 35, 158 priority levels, 63-70 priority queues, 127 processes, 3-4 lightweight, 5 processes, 3-4 lightweight, 5 program counter, 4 programs and programming complex example of, 201-212 definition, 3 event-driven input, 213-222 producer/consumer problem, 181-199 with threads, 7-29 writing simple multithread, 155-180 Q queues, 127-143 event, 213 FIFO, 127 LIFO, 127 message, 22-23, 275-284 priority, 127 types of, 127 R scheduling, 4 two-level, 6 semaphores event, 19-20, 91-104 multiple wait 105-126 PROG1-1.C, 72-75 PROG7-2.C, 75-78 PROG7-2.C, 75-78 PROG7-3.C, 78-81 PROG7-3.C, 78-81 PROG7-3.C, 78-81 PROG7-3.C, 78-81 PROG7-3.C, 78-81 PROG7-3.C, 78-81 PROG7-2.C, 75-78 PROG7-2.C, 75-78 PROG7-2.C, 75-78 PROG7-3.C, 78-81 PROG7-2.C, 75-78 PROG7-2.C, 75-78 PROG7-2.C, 75-78 PROG7-2.C, 75-78 PROG7-2.C, 75-78 PROG7-2.C, 75-78 PROG8-1.C, 92-94 PROG8-1.C, 92-94 PROG8-1.C, 92-94 PROG8-1.C, 106-109 PROG9-1.C, 106-109 PROG9-1.C, 106-109 PROG9-1.C, 106-109 PROG9-1.C, 106-109 PROG9-2.C, 109-113 PROG9-1.C, 106-109 PROG10-1.C, 128-130 PROG10-1.C, 166-107 PROG11-1.C, 146-147 PROG11-2.C, 159-162 PROG12-2.C,
paintRegion, 272 paintRegion, 272 paintRegionNum, 272 paintRegionNum, 272 paintRegionNum, 272 paintRegionNum, 272 paintRegionNum, 272 paintRegion, 272 paintRegion, 272 paintRegion, 272 paintRegion, 272 paintRegion, 272 program conder, 4-5 Presentation Manager message queues, 275-284 multithreaded painting, 255-274 using server threads under, 225-253 printf(), 35, 158 priority levels, 63-70 priority queues, 127 processes, 3-4 lightweight, 5 processEvent(), 222 program counter, 4 programs and programming complex example of, 201-212 definition, 3 event-driven input, 213-222 producer/consumer problem, 181-199 with threads, 7-29 writing simple multithread, 155-180 Q queues, 127-143 event, 213 PIFO, 127 LIFO, 127 message, 22-23, 275-284 priority, 127 types of, 127 R scheduling, 4 two-level, 6 semaphores event, 19-20, 91-104 multiple wait 105-196 PROG7-2.C, 75-78 PROG7-2.C, 75-78 PROG7-3.C, 78-81 PROG7-3.C, 78-81 PROG7-3.C, 78-81 PROG7-3.C, 78-81 PROG8-1.C, 92-94 PROG8-1.C, 02-94 PROG8-1.C, 02-94 PROG9-1.C, 106-109 PROG9-1.C, 106-109 PROG1-1.C, 128-130 PROG9-2.C, 109-113 PROG9-2.C, 109-113 PROG9-2.C, 109-113 PROG9-2.C, 109-113 PROG9-2.C, 106-109 PROG1-1.C, 128-130 PROG1-2.C, 130-133 PROG1-2.C, 150-162 PROG1-2.C, 150-162 PROG1-2.C, 150-162 PROG12-2.C, 159-162 PROG12-2.C, 159-162 PROG12-3.C, 163-166 PROG1-2.C, 163-166 PROG1-2.C, 177-180 PROG13-2.C, 183-184 PROG13-2.C, 183-184 PROG13-2.C, 183-18 PROG13-2.C, 183-18 PROG13-2.C, 183-18 PROG13-2.C, 183-18 PROG13-2.C, 183-18 PROG12-3.C, 163-16 PROG1-1.C, 128-187 PROG13-2.C, 183-18 PROG12-1.C, 166-157 PROG12-2.C, 159-162 PROG12-3.C, 163-166 PROG1-2.C, 203-162 PROG13-2.C, 183-18 PROG
paintRegionNum, 272 paintSignal, 272, 273, 274 preemptive scheduler, 4-5 Presentation Manager message queues, 275-284 militihreaded painting, 255-274 using server threads under, 225-253 printf(), 35, 158 priority levels, 63-70 priority queues, 127 processes, 3-4 lightweight, 5 processes, 3-4 lightweight, 5 processes weightly, 222 program counter, 4 programs and programming complex example of, 201-212 definition, 3 event-driven input, 213-222 producer/consumer problem, 181-199 with threads, 7-29 writing simple multithread, 155-180 Q queues, 127-143 event, 213 FIFO, 127 LIFO, 127 message, 22-23, 275-284 priority, 127 types of, 127 R scheduling, 4 two-level, 6 semaphores event, 19-20, 91-104 multiple wait 105-126 PROG7-3.C, 75-78 PROG7-3.C, 78-81 PROG6-1.C, 228-94 PROG8-1.C, 106-109 PROG9-1.C, 106-109 PROG9-1.C, 106-109 PROG9-1.C, 106-109 PROG9-1.C, 106-109 PROG1-1.C, 128-130 PROG10-1.C, 128-130 PROG10-1.C, 128-130 PROG10-1.C, 128-130 PROG10-1.C, 128-130 PROG10-1.C, 128-130 PROG1-2.C, 109-113 PROG10-1.C, 128-130 PROG10-1.C, 128-1
paintSignal, 272, 273, 274 preemptive scheduler, 4-5 Presentation Manager message queues, 275-284 multithreaded painting, 255-274 using server threads under, 225-253 printf(), 35, 158 priority levels, 63-70 priority queues, 127 processes, 3-4 lightweight, 5 processEvent(), 222 program counter, 4 programs and programming complex example of, 201-212 definition, 3 event-driven input, 213-222 producer/consumer problem, 181-199 with threads, 7-29 writing simple multithread, 155-180 Q queues, 127-143 event, 213 FIFO, 127 LIFO, 127 message, 22-23, 275-284 priority, 127 types of, 127 R scheduling, 4 two-level, 6 semaphores event, 19-20, 91-104 multithe wait 105-126 PROG3-1.C, 106-109 PROG3-2.C, 109-113 PROG9-3.C, 113-116 PROG9-3.C, 130-133 PROG10-1.C, 128-130 PROG10-1.C, 128-130 PROG10-1.C, 126-16 PROG12-2.C, 130-133 PROG10-1.C, 145-147 PROG12-2.C, 159-162 PROG12-2.C, 169-162 PROG12-3.C, 163-166 PROG12-4.C, 167-170 PROG12-5.C, 159-162 PROG12-3.C, 163-166 PROG12-4.C, 167-170 PROG12-5.C, 189-193 PROG13-1.C, 182-187 PROG13-1.C, 182-187 PROG13-1.C, 182-187 PROG13-1.C, 214-222 PROG14-1.C, 201-211 PROG15-1.C, 214-222 PROG16-1.R, 236-238 PROG16-1.R, 236-238 PROG16-1.R, 236-238 PROG16-1.R, 236-238 PROG16-1.R, 236-238 PROG16-1.C, 226-236 PROG13-1.C, 262-236 PROG13-1.C, 177-180 PROG13-2.C, 130-133 PROG13-1.C, 185-147 PROG13-2.C, 130-133 PROG13-1.C, 185-147 PROG13-2.C, 130-133 PROG13-1.C, 185-147 PROG13-2.C, 130-134 P
preemptive scheduler, 4-5 Presentation Manager message queues, 275-284 multithreaded painting, 255-274 using server threads under, 225-253 printf(), 35, 158 priority levels, 63-70 priority queues, 127 processes, 3-4 lightweight, 5 program counter, 4 programs and programming complex example of, 201-212 definition, 3 event-driven input, 213-222 producer/consumer problem, 181-199 with threads, 7-29 writing simple multithread, 155-180 Q queues, 127-143 event, 213 FIFO, 127 message, 22-23, 275-284 priority, 127 types of, 127 R race condition, 10 repeated-interval timer, 145 ringBufferEmpty, 188 ringBufferFull, 188 S scheduling, 4 two-level, 6 semaphores event, 19-20, 91-104 multiple wait 105-126 PROG8-1.C, 192-19 PROG8-2.C, 94-97 PROG9-2.C, 109-113 PROG9-2.C, 109-113 PROG9-3.C, 113-116 PROG10-1.C, 128-130 PROG10-1.C, 126-130 PROG10-2.C, 130-133 PROG10-1.C, 145-147 PROG11-2.C, 147-149 PROG11-2.C, 156-157 PROG12-3.C, 163-166 PROG12-4.C, 167-170 PROG12-4.C, 167-170 PROG12-4.C, 167-170 PROG12-4.C, 167-170 PROG12-4.C, 167-170 PROG12-5.C, 172-176 PROG13-1.C, 182-187 PROG13-1.C, 182-187 PROG13-1.C, 214-222 PROG13-1.C, 214-222 PROG13-1.C, 214-222 PROG16-1.C, 226-236 PROG16-1.C, 239-252 PROG17-1.C, 256-270 PROG18-1.C, 276-283 strlen(), 36 T thread execution, suspending and resuming, 55 62 thread rescheduling, enabling and disabling, 49-53 changing priority of, 63-70 programming with, 7-29
message queues, 275-284 multithreaded painting, 255-274 using server threads under, 225-253 printf(), 35, 158 priority levels, 63-70 priority queues, 127 processes, 3-4 lightweight, 5 processes, 3-4 lightweight, 5 program counter, 4 programs and programming complex example of, 201-212 definition, 3 event-driven input, 213-222 producer/consumer problem, 181-199 with threads, 7-29 writing simple multithread, 155-180 Q queues, 127-143 event, 213 FIFO, 127 LIFO, 127 message, 22-23, 275-284 priority, 127 types of, 127 R race condition, 10 repeated-interval timer, 145 ringBufferEmpty, 188 ringBufferFull, 188 S scheduling, 4 two-level, 6 semaphores event, 19-20, 91-104 multithreads under, 225-253 PROG8-2.C, 94-97 PROG93-1.C, 106-109 PROG9-3.C, 109-113 PROG9-3.C, 113-116 PROG10-1.C, 128-130 PROG11-1.C, 145-147 PROG11-1.C, 145-147 PROG11-2.C, 147-149 PROG11-2.C, 147-149 PROG12-3.C, 163-166 PROG12-4.C, 167-170 PROG12-3.C, 163-166 PROG12-4.C, 167-170 PROG12-3.C, 189-192 PROG13-1.C, 182-187 PROG13-1.C, 182-187 PROG13-1.C, 214-222 PROG16-1.C, 226-236 PROG16-1.C, 226-236 PROG16-1.C, 226-236 PROG16-1.C, 226-236 PROG16-1.C, 226-236 PROG16-1.C, 256-270 PROG18-1.C, 256-270 PROG18-2.C, 39-252 PROG18-1.C, 256-270 PROG18-2.C, 239-252 PROG18-1.C, 256-270 PROG18-2.C, 163-166 PROG18-2.C, 163-166 PROG18-2.C
multithreaded painting, 255-274 using server threads under, 225-253 printf(), 35, 158 priority levels, 63-70 priority queues, 127 processes, 3-4 lightweight, 5 processEvent(), 222 program counter, 4 programs and programming complex example of, 201-212 definition, 3 event-driven input, 213-222 producer/consumer problem, 181-199 with threads, 7-29 writing simple multithread, 155-180 Q queues, 127-143 event, 213 FIFO, 127 LIFO, 127 message, 22-23, 275-284 priority, 127 types of, 127 R race condition, 10 repeated-interval timer, 145 ringBufferEmpty, 188 ringBufferFull, 188 S scheduling, 4 two-level, 6 semaphores event, 19-20, 91-104 multiple wait 105-196 PROG1-1.C, 106-109 PROG9-1.C, 106-109 PROG9-2.C, 109-113 PROG9-3.C, 113-116 PROG10-2.C, 130-133 PROG10-2.C, 130-133 PROG10-2.C, 130-133 PROG11-1.C, 145-147 PROG11-2.C, 147-149 PROG11-2.C, 147-149 PROG12-3.C, 163-166 PROG12-4.C, 167-170 PROG12-6.C, 177-180 PROG13-1.C, 126-26-26 PROG13-1.C, 126-25 PROG13-1.C, 126-25 PROG16-1.C, 226-236 PROG18-1.C, 226-236 PROG16-1.C, 226-236 PROG18-1.C, 226-236 PROG16-1.C, 226-236 PROG18-1.C, 226-236 PROG18-
using server threads under, 225-253 printf(), 35, 158 priority levels, 63-70 priority queues, 127 processes, 3-4 lightweight, 5 program counter, 4 programs and programming complex example of, 201-212 definition, 3 event-driven input, 213-222 producer/consumer problem, 181-199 with threads, 7-29 writing simple multithread, 155-180 Q queues, 127-143 event, 213 FIFO, 127 LIFO, 127 message, 22-23, 275-284 priority, 127 types of, 127 R race condition, 10 repeated-interval timer, 145 ringBufferEmpty, 188 ringBufferFull, 188 S scheduling, 4 two-level, 6 semaphores event, 19-20, 91-104 multiple wait 105-196
printf(), 35, 158 priority levels, 63-70 priority queues, 127 processes, 3-4 lightweight, 5 processes, 3-4 lightweight, 5 program counter, 4 programs and programming complex example of, 201-212 definition, 3 event-driven input, 213-222 producer/consumer problem, 181-199 with threads, 7-29 with threads, 7-29 writing simple multithread, 155-180 Q queues, 127-143 event, 213 FIFO, 127 LIFO, 127 LIFO, 127 message, 22-23, 275-284 priority, 127 types of, 127 R race condition, 10 repeated-interval timer, 145 ringBufferFmpty, 188 ringBufferFmptyt, 188 ringBufferFmptyt, 188 ringBufferFmptyt, 188 ringBufferFmptyt, 188 ringBufferFmptyt, 188 ringBufferFull, 188 S scheduling, 4 two-level, 6 semaphores event, 19-20, 91-104 multiple wait 105-196
priority levels, 63-70 priority queues, 127 processes, 3-4 lightweight, 5 processEvent(), 222 program counter, 4 programs and programming complex example of, 201-212 definition, 3 event-driven input, 213-222 producer/consumer problem, 181-199 with threads, 7-29 writing simple multithread, 155-180 Q queues, 127-143 event, 213 Event, 213 FIFO, 127 LIFO, 127 message, 22-23, 275-284 priority, 127 types of, 127 R race condition, 10 repeated-interval timer, 145 ringBufferEmpty, 188 ringBufferFull, 188 S scheduling, 4 two-level, 6 semaphores event, 19-20, 91-104 multiple wait 105-126 PROG10-1. C, 128-130 PROG10-1. C, 130-133 PROG11-2. C, 147-149 PROG12-2. C, 159-162 PROG12-3. C, 163-166 PROG12-4. C, 167-170 PROG12-5. C, 172-176 PROG12-6. C, 177-180 PROG13-1. C, 182-187 PROG13-1. C, 182-187 PROG13-1. C, 182-187 PROG13-1. C, 214-219 PROG16-1. C, 214-222 PROG16-1. C, 214-222 PROG16-1. C, 226-236 PROG16-1. R, 236-238 PROG16-1. R, 236-238 PROG16-1. R, 236-238 PROG16-1. R, 236-238 PROG18-1. C, 276-283 strlen(), 36 synchronization, 16, 166, 170, 212 T thread execution, suspending and resuming, 55 62 thread rescheduling, enabling and disabling, 49-53 threadRunningSem(), 171, 176 threads, 5 changing priority of, 63-70 programming with, 7-29
priority queues, 127 processes, 3-4 lightweight, 5 processEvent(), 222 program counter, 4 programs and programming complex example of, 201-212 definition, 3 event-driven input, 213-222 producer/consumer problem, 181-199 with threads, 7-29 writing simple multithread, 155-180 Q queues, 127-143 event, 213 FIFO, 127 message, 22-23, 275-284 priority, 127 types of, 127 R R scheduling, 4 two-level, 6 semaphores event, 19-20, 91-104 multiple wait 105-126 PROG10-2.C, 130-183 PROG11-1.C, 145-147 PROG11-2.C, 147-149 PROG12-2.C, 159-162 PROG12-4.C, 167-170 PROG12-4.C, 167-170 PROG12-5.C, 172-176 PROG12-5.C, 172-176 PROG13-1.C, 182-187 PROG13-1.C, 182-187 PROG13-1.C, 182-187 PROG13-1.C, 201-211 PROG15-1.C, 214-222 PROG16-1.C, 226-236 PROG16-1.C, 226-236 PROG16-1.R, 238 PROG16-1.RC, 238 PROG16-1.RC, 239-252 PROG16-1.RC, 239-252 PROG16-1.RC, 239-252 PROG16-1.RC, 238 strlen(), 36 synchronization, 16, 166, 170, 212 T thread execution, suspending and resuming, 55 62 thread rescheduling, enabling and disabling, 49-53 threadRunningSem(), 171, 176 threads, 5 changing priority of, 63-70 programming with, 7-29
processes, 3-4 lightweight, 5 processeEvent(), 222 program counter, 4 programs and programming complex example of, 201-212 definition, 3 event-driven input, 213-222 producer/consumer problem, 181-199 with threads, 7-29 writing simple multithread, 155-180 Q queues, 127-143 event, 213 FIFO, 127 message, 22-23, 275-284 priority, 127 types of, 127 R scheduling, 4 two-level, 6 semaphores event, 19-20, 91-104 multiple wait 105-126 PROG11-1.C, 147-149 PROG12-1.C, 156-157 PROG12-2.C, 159-162 PROG12-3.C, 163-166 PROG12-4.C, 167-170 PROG12-5.C, 172-176 PROG12-5.C, 172-176 PROG13-1.C, 182-187 PROG13-1.C, 182-187 PROG13-1.C, 182-187 PROG13-1.C, 182-187 PROG13-1.C, 214-199 PROG13-1.C, 214-199 PROG13-1.C, 214-199 PROG13-1.C, 214-199 PROG13-1.C, 214-199 PROG13-1.C, 216-236 PROG13-1.C, 216-236 PROG13-1.C, 216-236 PROG13-1.C, 216-283 Stringlater PROG13-1.C, 238-193 PROG13-1.C, 216-283 PROG13-1.C, 216-2863 PROG13-1.C, 216-2863 PROG13-1.C, 216-2863 PROG13-1.C, 216-2863 PROG13-1.C, 216-2863 PROG13-1.C, 216-2863 PROG13-1.C, 216-287 PROG13-1.C, 216-
lightweight, 5 processEvent(), 222 program counter, 4 programs and programming complex example of, 201-212 definition, 3 event-driven input, 213-222 producer/consumer problem, 181-199 with threads, 7-29 writing simple multithread, 155-180 Q queues, 127-143 event, 213 FIFO, 127 LIFO, 127 message, 22-23, 275-284 priority, 127 types of, 127 R race condition, 10 repeated-interval timer, 145 ringBufferEmpty, 188 ringBufferEmpty, 188 ringBufferFull, 188 S scheduling, 4 two-level, 6 semaphores event, 19-20, 91-104 multiple wait, 105-126 PROG12-1.C, 156-157 PROG12-2.C, 163-166 PROG12-4.C, 167-170 PROG12-5.C, 172-176 PROG12-5.C, 177-180 PROG13-1.C, 182-187 PROG13-1.C, 182-187 PROG13-1.C, 189-193 PROG13-1.C, 189-193 PROG13-1.C, 189-193 PROG13-1.C, 189-193 PROG13-1.C, 180-193 PROG13-1.C, 214-222 PROG13-1.C, 214-222 PROG16-1.C, 226-236 PROG16-1.C, 226-236 PROG16-1.C, 226-236 PROG16-1.R, 236-238 PROG16-1.C, 226-236 PROG18-1.C, 276-283 strlen(), 36 synchronization, 16, 166, 170, 212 T thread execution, suspending and resuming, 55 62 thread rescheduling, enabling and disabling, 49-53 threadRunningSem(), 171, 176 threads, 5 changing priority of, 63-70 programming with, 7-29
processEvent(), 222 program counter, 4 programs and programming complex example of, 201-212 definition, 3 event-driven input, 213-222 producer/consumer problem, 181-199 with threads, 7-29 writing simple multithread, 155-180 Q queues, 127-143 event, 213 FIFO, 127 message, 22-23, 275-284 priority, 127 types of, 127 R race condition, 10 repeated-interval timer, 145 ringBufferFull, 188 S scheduling, 4 two-level, 6 semaphores event, 19-20, 91-104 multiple wait 105-126 PROG12-2. C, 159-162 PROG12-3. C, 163-166 PROG12-4. C, 167-170 PROG12-5. C, 172-176 PROG12-5. C, 177-180 PROG13-1. C, 182-187 PROG13-2. C, 189-193 PROG13-3. C, 194-199 PROG13-1. C, 201-211 PROG13-1. C, 201-236 PROG16-1. R. C, 238 PROG16-1. R. C
program counter, 4 programs and programming
complex example of, 201-212 definition, 3 event-driven input, 213-222 producer/consumer problem, 181-199 with threads, 7-29 writing simple multithread, 155-180 Q queues, 127-143 event, 213 FIFO, 127 message, 22-23, 275-284 priority, 127 types of, 127 R race condition, 10 repeated-interval timer, 145 ringBufferFull, 188 S scheduling, 4 two-level, 6 semaphores event, 19-20, 91-104 multiple wait, 105-126 PROG12-4.C, 167-170 PROG12-5.C, 172-176 PROG12-6.C, 177-180 PROG13-1.C, 182-187 PROG13-2.C, 189-193 PROG13-3.C, 194-199 PROG13-1.C, 201-211 PROG15-1.C, 214-222 PROG16-1.R, 236-236 PROG16-1.R, 236-238 PROG16-1.RC, 238 PROG16-1.RC, 238 PROG16-1.RC, 238 PROG16-1.RC, 238 Synchronization, 16, 166, 170, 212 T T thread execution, suspending and resuming, 55 62 thread rescheduling, enabling and disabling, 49-53 threadRunningSem(), 171, 176 threads, 5 changing priority of, 63-70 programming with, 7-29
definition, 3 event-driven input, 213-222 producer/consumer problem, 181-199 with threads, 7-29 writing simple multithread, 155-180 Q Q PROG12-5.C, 172-176 PROG12-6.C, 177-180 PROG13-1.C, 182-187 PROG13-1.C, 182-187 PROG13-2.C, 189-193 PROG13-3.C, 194-199 PROG13-3.C, 194-199 PROG14-1.C, 201-211 PROG15-1.C, 214-222 Queues, 127-143 event, 213 FIFO, 127 PROG16-1.C, 226-236 PROG16-1.R, 236-238 FIFO, 127 PROG16-1.R, 236-238 PROG16-1.RC, 238 PROG16-1.RC, 238 PROG16-1.RC, 238 PROG16-1.C, 256-270 PROG18-1.C, 276-283 strlen(), 36 synchronization, 16, 166, 170, 212 T T T T T triang Lifer Empty, 188 ring Buffer Empty, 188 ring Buffer Empty, 188 ring Buffer Empty, 188 ring Buffer Empty, 188 thread execution, suspending and resuming, 55 62 thread rescheduling, enabling and disabling, 49-53 scheduling, 4 two-level, 6 semaphores event, 19-20, 91-104 multiple wait 105-126 T thread Running Sem(), 171, 176 threadS, 5 changing priority of, 63-70 programming with, 7-29
event-driven input, 213-222 producer/consumer problem, 181-199 with threads, 7-29 writing simple multithread, 155-180 Q queues, 127-143 event, 213 FIFO, 127 LIFO, 127 message, 22-23, 275-284 priority, 127 types of, 127 R race condition, 10 repeated-interval timer, 145 ringBufferEmpty, 188 ringBufferFull, 188 S scheduling, 4 two-level, 6 semaphores event, 19-20, 91-104 multiple wait, 105-126 PROG12-6.C, 177-180 PROG13-1.C, 182-187 PROG13-2.C, 189-193 PROG13-3.C, 194-199 PROG13-3.C, 194-199 PROG16-1.C, 201-211 PROG15-1.C, 214-222 PROG16-1.C, 226-236 PROG16-1.R, 238 PROG16-1.R, 236 PROG16-1.R, 236 Synchronization, 16, 166, 170, 212 T thread execution, suspending and resuming, 55 62 thread rescheduling, enabling and disabling, 49-53 threadRunningSem(), 171, 176 threads, 5 changing priority of, 63-70 programming with, 7-29
producer/consumer problem, 181-199 with threads, 7-29 writing simple multithread, 155-180 Q queues, 127-143 event, 213 FIFO, 127 LIFO, 127 message, 22-23, 275-284 priority, 127 types of, 127 R race condition, 10 repeated-interval timer, 145 ringBufferEmpty, 188 ringBufferFull, 188 S scheduling, 4 two-level, 6 semaphores event, 19-20, 91-104 multiple wait, 105-126 PROG13-1.C, 182-187 PROG13-2.C, 189-193 PROG13-3.C, 194-199 PROG13-1.C, 201-211 PROG15-1.C, 214-222 PROG16-1.C, 226-236 PROG16-1.RC, 238 PROG16-1.RC, 238 PROG16-1.RC, 238 PROG16-1.RC, 239-252 PROG17-1.C, 256-270 PROG18-1.C, 276-283 strlen(), 36 synchronization, 16, 166, 170, 212 T thread execution, suspending and resuming, 55 62 thread rescheduling, enabling and disabling, 49-53 threadRunningSem(), 171, 176 threads, 5 changing priority of, 63-70 programming with, 7-29
with threads, 7-29 writing simple multithread, 155-180 Q queues, 127-143 event, 213 FIFO, 127 message, 22-23, 275-284 priority, 127 types of, 127 R race condition, 10 repeated-interval timer, 145 ringBufferFull, 188 S scheduling, 4 two-level, 6 semaphores event, 19-20, 91-104 multiple wait, 105-126 PROG13-2.C, 189-193 PROG13-2.C, 194-199 PROG14-1.C, 201-211 PROG15-1.C, 226-236 PROG16-1.H, 236-238 PROG16-1.K, 26-238 PROG16-1.K, 236-238 PRO
writing simple multithread, 155-180 Q queues, 127-143
Q
queues, 127-143 event, 213 PROG16-1.C, 226-236 PROG16-1.H, 236-238 FIFO, 127 PROG16-1.RC, 238 LIFO, 127 PROG16-1.RC, 238 PROG16-1.RC, 238 PROG16-1.RC, 238 PROG16-1.RC, 239-252 PROG16-1.RC, 239-252 PROG16-1.RC, 239-252 PROG16-1.RC, 239-252 PROG17-1.C, 256-270 PROG18-1.C, 276-283 strlen(), 36 synchronization, 16, 166, 170, 212 R race condition, 10 repeated-interval timer, 145 ringBufferEmpty, 188 ringBufferFull, 188 S scheduling, 4 two-level, 6 semaphores event, 19-20, 91-104 multiple wait 105-126 PROG16-1.RC, 226-236 PROG16-1.RC, 236-238 PROG16-1.RC, 238 PROG16-1.RC, 239-252 PROG16-1.RC, 239-252 PROG16-1.RC, 238 PROG16-1.RC, 238 PROG16-1.RC, 238 PROG16-1.RC, 239-252 PROG16-1.RC, 239-2
event, 213 FIFO, 127 FIFO, 127 PROG16-1. II, 236-238 FIFO, 127 PROG16-1. RC, 238 LIFO, 127 PROG16-2. C, 239-252 PROG17-1. C, 256-270 PROG18-1. C, 276-283 strlen(), 36 synchronization, 16, 166, 170, 212 R race condition, 10 repeated-interval timer, 145 ringBufferEmpty, 188 ringBufferFull, 188 S S S S Scheduling, 4 two-level, 6 semaphores event, 19-20, 91-104 multiple wait 105-126 PROG16-1. II, 236-238 PROG16-1. II, 236-24
FIFO, 127 LIFO, 127 message, 22-23, 275-284 priority, 127 types of, 127 Race condition, 10 repeated-interval timer, 145 ringBufferEmpty, 188 ringBufferFull, 188 S S Scheduling, 4 two-level, 6 semaphores event, 19-20, 91-104 multiple wait, 105-126 PROG16-1.RC, 238 PROG16-1.RC, 239 PROG16-1.RC, 236 PROG18-1.C, 256 P
LIFO, 127 message, 22-23, 275-284 priority, 127 types of, 127 R race condition, 10 repeated-interval timer, 145 ringBufferEmpty, 188 ringBufferFull, 188 S scheduling, 4 two-level, 6 semaphores event, 19-20, 91-104 multiple wait, 105-126 PROG16-2.C, 239-252 PROG17-1.C, 256-270 PROG18-1.C, 276-283 strlen(), 36 synchronization, 16, 166, 170, 212 T thread execution, suspending and resuming, 55 62 thread rescheduling, enabling and disabling, 49-53 threadRunningSem(), 171, 176 threads, 5 changing priority of, 63-70 programming with, 7-29
message, 22-23, 275-284 priority, 127 types of, 127 R strlen(), 36 synchronization, 16, 166, 170, 212 T race condition, 10 repeated-interval timer, 145 ringBufferEmpty, 188 ringBufferFull, 188 S scheduling, 4 two-level, 6 semaphores event, 19-20, 91-104 multiple wait, 105-126 PROG17-1.C, 256-270 PROG18-1.C, 276-283 strlen(), 36 synchronization, 16, 166, 170, 212 T thread execution, suspending and resuming, 55 62 thread rescheduling, enabling and disabling, 49- 53 threadRunningSem(), 171, 176 threads, 5 changing priority of, 63-70 programming with, 7-29
types of, 127 R race condition, 10 repeated-interval timer, 145 ringBufferEmpty, 188 ringBufferFull, 188 S scheduling, 4 two-level, 6 semaphores event, 19-20, 91-104 multiple wait 105-126 race condition, 10 repeated-interval timer, 145 ringBufferEmpty, 188 thread execution, suspending and resuming, 55 62 thread rescheduling, enabling and disabling, 49- 53 threadRunningSem(), 171, 176 threads, 5 changing priority of, 63-70 programming with, 7-29
R race condition, 10 repeated-interval timer, 145 ringBufferEmpty, 188 ringBufferFull, 188 S scheduling, 4 two-level, 6 semaphores event, 19-20, 91-104 multiple wait 105-126 race condition, 16, 166, 170, 212 T thread execution, suspending and resuming, 55 62 thread rescheduling, enabling and disabling, 49- 53 threadRunningSem(), 171, 176 threads, 5 changing priority of, 63-70 programming with, 7-29
race condition, 10 repeated-interval timer, 145 ringBufferEmpty, 188 ringBufferFull, 188 S scheduling, 4 two-level, 6 semaphores event, 19-20, 91-104 multiple wait 105-126 T thread execution, suspending and resuming, 55 62 thread rescheduling, enabling and disabling, 49- 53 threadRunningSem(), 171, 176 threads, 5 changing priority of, 63-70 programming with, 7-29
repeated-interval timer, 145 ringBufferEmpty, 188 ringBufferFull, 188 S scheduling, 4 two-level, 6 semaphores event, 19-20, 91-104 multiple wait 105-126 T thread execution, suspending and resuming, 55 62 thread rescheduling, enabling and disabling, 49- 53 threadRunningSem(), 171, 176 threads, 5 changing priority of, 63-70 programming with, 7-29
ringBufferEmpty, 188 ringBufferFull, 188 S scheduling, 4 two-level, 6 semaphores event, 19-20, 91-104 multiple wait 105-126 thread execution, suspending and resuming, 55 62 thread rescheduling, enabling and disabling, 49- 53 threadRunningSem(), 171, 176 threads, 5 changing priority of, 63-70 programming with, 7-29
ringBufferFull, 188 S scheduling, 4 two-level, 6 semaphores event, 19-20, 91-104 multiple wait 105-126 thread rescheduling, enabling and disabling, 49-53 threadRunningSem(), 171, 176 threads, 5 changing priority of, 63-70 programming with, 7-29
S thread rescheduling, enabling and disabling, 49- scheduling, 4 two-level, 6 semaphores event, 19-20, 91-104 multiple wait 105-126 threadRunningSem(), 171, 176 threads, 5 changing priority of, 63-70 programming with, 7-29
scheduling, 4 two-level, 6 semaphores event, 19-20, 91-104 multiple wait 105-126 scheduling, 4 threadRunningSem(), 171, 176 threads, 5 changing priority of, 63-70 programming with, 7-29
two-level, 6 threadRunningSem(), 171, 176 semaphores threads, 5 event, 19-20, 91-104 changing priority of, 63-70 multiple wait 105-126 programming with, 7-29
semaphores threads, 5 event, 19-20, 91-104 changing priority of, 63-70 multiple wait 105-126 programming with, 7-29
event, 19-20, 91-104 changing priority of, 63-70 programming with, 7-29
multiple wait 105-126 programming with, 7-29
mutual exclusion 13 71-90 server, 201, 211, 225-253
server threads 201 211 225-253 Starting and ending, 33-48
single-interval timer 145 unlead walting belief (), 102, 100, 170
source code timers, 145-152 repeated-interval, 145
r NOG2-1.0, 6-9 single-interval 145
r 10 timing bug 10
PROG2-3.C, 14-16 PROG2-4.C, 17-19 two-level scheduling, 6
PROG2-5.C, 20-22 W
PROG2-6.C, 23-25 WinCreateMsgQueue(), 252
PROG2-7.C, 26-28 WinQueryWindowPtr(), 276
PROG3-1.C, 34-35 WinSetWindowPtr(), 276

About the Authors

Len Dorfman has been an educator in the Long Island, New York, school system since 1970. A programmer with extensive experience with C and Assembler, he is the author or coauthor of $OS/2^{\circ}$ Extra! KBD, MOU, and VIO Special Functions Revealed (McGraw-Hill, 1993); Effective OS/2 Multithreading (McGraw-Hill, 1994); and C Memory Management Techniques (McGraw-Hill, 1993). He holds a Ph.D. in education from Hofstra University.

Marc Neurberger holds a Ph.D. in Computer Science from the State University of New York at Stony Brook. He is a Software Engineer at Dataware Technologies in Cambridge, Massachusetts. Together with Len Dorfman, he has written several books, including C^{++} Memory Management Techniques.



DISK WARRANTY

This software is protected by both United States copyright law and international copyright treaty provision. You must treat this software just like a book, except that you may copy it into a computer in order to be used and you may make archival copies of the software for the sole purpose of backing up our software and protecting your investment from loss.

By saying "just like a book," McGraw-Hill means, for example, that this software may be used by any number of people and may be freely moved from one computer location to another, so long as there is no possibility of its being used at one location or on one computer while it also is being used at another. Just as a book cannot be read by two different people in two different places at the same time, neither can the software be used by two different people in two different places at the same time (unless, of course, McGraw-Hill's copyright is being violated).

LIMITED WARRANTY

Windcrest/McGraw-Hill takes great care to provide you with top-quality software, thoroughly checked to prevent virus infections. McGraw-Hill warrants the physical diskette(s) contained herein to be free of defects in materials and workmanship for a period of sixty days from the purchase date. If McGraw-Hill receives written notification within the warranty period of defects in materials or workmanship, and such notification is determined by McGraw-Hill to be correct, McGraw-Hill will replace the defective diskette(s). Send requests to:

Customer Service Windcrest/McGraw-Hill 13311 Monterey Lane Blue Ridge Summit, PA 17294-0850

The entire and exclusive liability and remedy for breach of this Limited Warranty shall be limited to replacement of defective diskette(s) and shall not include or extend to any claim for or right to cover any other damages, including but not limited to, loss of profit, data, or use of the software, or special, incidental, or consequential damages or other similar claims, even if McGraw-Hill has been specifically advised of the possibility of such damages. In no event will McGraw-Hill's liability for any damages to you or any other person ever exceed the lower of suggested list price or actual price paid for the license to use the software, regardless of any form of the claim.

McGRAW-HILL, INC. SPECIFICALLY DISCLAIMS ALL OTHER WARRANTIES, EXPRESS OR IMPLIED, INCLUDING, BUT NOT LIMITED TO, ANY IMPLIED WARRANTY OF MERCHANTABILITY OR FITNESS FOR A PARTICULAR PURPOSE.

Specifically, McGraw-Hill makes no representation or warranty that the software is fit for any particular purpose and any implied warranty of merchantability is limited to the sixty-day duration of the Limited Warranty covering the physical diskette(s) only (and not the software) and is otherwise expressly and specifically disclaimed.

This limited warranty gives you specific legal rights; you may have others which may vary from state to state. Some states do not allow the exclusion of incidental or consequential damages, or the limitation on how long an implied warranty lasts, so some of the above may not apply to you.

The enclosed high-density 3%" disk contains the self-extracting file, 4440DISK.EXE. This file contains 149 other programs in a compressed format.

Before you can access the programs contained in 4440DISK.EXE, you must create a directory on your hard drive, copy 4440DISK.EXE into the new directory, and run the 4440DISK.EXE file. (*Note:* You will need a 3.1Mb of available hard drive space to successfully decompress all of the files contained in 4440DISK.EXE.) At the C> prompt, type:

MD C:\directory_name

(where directory_name is the name of the directory that you want to create) and press Enter. You have just created a directory on your hard drive to hold all of the programs contained in 4440DISK.EXE. Now, place the companion diskette in drive A: and copy 4440DISK.EXE to your newly created directory by typing:

COPY A:\4440DISK.EXE C:\directory_name

(where directory_name is the name of the directory that you created earlier) and press Enter. Now, you can safely run 4440DISK.EXE to extract the other programs.

Change to the directory that contains 4440DISK.EXE by typing:

CD directory_name

at the C> prompt. To extract the file, type:

4440DISK

and press Enter. The program will list the name of each file as it is uncompressed and will return to the DOS prompt when all of the files have been uncompressed.

IMPORTANT

Read the Disk Warranty terms on the previous page before opening the disk envelope. Opening the envelop constitutes acceptance of these terms and renders this entire book-disk package nonreturnable except for placement in kind due to material defects.

Write effective multithreaded programs in C for O\$/2



EFFECTIVE MULTITHREADING IN OS/2°

If you are a C, Windows, or systems programmer ready to unlock the powerful multitasking capabilities of IBM's hot OS/2 2.1 operating system, here is your key. This is the *only* OS/2 book devoted *exclusively* to multithreading, presenting all the tools and the ready-to-use source code you'll need to successfully integrate your application program with OS/2.

Beginning with the basics of writing multithreaded programs and an overview of the massive OS/2 operating system, *Effective Multithreading* then moves to more complex demonstration programs and presents the OS/2 programmer's interface. The book then covers topics such as threads, priorities, scheduling semaphores, queues, timers, and much more—everything you need for complete understanding! What's more, each topic is thoroughly explained and punctuated with heavily documented C source code.

With more than 20 fully documented programs presented and an accompanying disk with complete program listings for all the source code, *Effective Multithreading* is a must-have for all programmers wishing to tap into OS/2's diversity.





er: Holberg Design

Graw-Hill, Inc.

ie Need for Knowledge
enue of the Americas
New York, NY 10020